Ausarbeitung zum Thema AVL Bäume und deren Implementierung in Erlang

Algorithmen und Datenstrukturen, WiSe 17/18

Enes Kaya

13. Dezember, 2017

Referat eingereicht im Rahmen der Vorlesung Algorithmen und Datenstrukturen
im Studiengang Angewandte Informatik (AI)
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. C. Klauck
Abgegeben am 12.12.2017

Inhaltsverzeichnis

1 Erklärung zur schriftlichen Ausarbeitung des Referates

Hiermit erkläre ich, dass ich diese schriftliche Ausarbeitung meines Referates selbstständig und ohne fremde Hilfe verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe sowie die aus fremden Quellen (dazu zählen auch Internetquellen) direkt oder indirekt übernommenen Gedanken oder Wortlaute als solche kenntlich gemacht habe. Zudem erkläre ich, dass der zugehörige Programmcode von mir selbständig implementiert wurde ohne diesen oder Teile davon von Dritten im Wortlaut oder dem Sinn nach übernommen zu haben. Die Arbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher nicht veröffentlicht.

Hamburg, den 12.12.2017

2 Aufgabenbeschreibung

Zu implementieren sei der abstrakte Datentyp AVL-Baum in der funktionalen Sprache Erlang. AVL Bäume sind höhen-balancierte Binär-Bäume. Durch diese Daten-Struktur, die zum Beispiel in Datenbanksystemen im Hintergrund für einen effizienten Zugriff auf Schlüssel gewährleistet, verhindert man u.a. dass Bäume bspw. zu Listen werden und so die erwartet Effizienz von Einfüge-, Lösch- und Such-Operationen verschlechtern. Das Suchen in einem AVL-Baum hat den Aufwand $O(log(n))$, dies wird später in den Tests und den Experimenten gezeigt. Diese Daten-Struktur wurde bereits 1962 von **A**delson-**V**elskii und **L**andis vorgeschlagen (Daher auch der Name AVL).

2.1 Geforderte Operationen

Um mit der Datenstruktur zu arbeiten werden folgende Operationen gefordert und implementiert:

1. $initBT()$: Gibt den initialen Baum zurück und dient als Konstruktor.
2. $isEmptyBT(B)$: Prüft ob ein gegebener Baum B leer ist und liefert einen Boolean
3. $equalBT(B_1, B_2)$: Prüft rekursiv ob zwei gegebene Bäume strukturgleich sind und liefert einen Boolean
4. $isBT(B)$: Prüft ob gegebener Baum B ein gültiger AVL-Baum ist und liefert einen Boolean
5. $insertBT(B, N)$: Fügt rekursiv das Element N in B ein und liefert einen neuen Baum inklusive dem Element N. Führt eventuelle Rotations-Operationen aus.
6. $deleteBT(B, N)$: Löscht rekursiv das Element N aus B und liefert einen neuen Baum exklusive dem Element N. Führt eventuelle Rotations-Operationen aus.
7. $printBT(B, Filename)$: Erstellt eine Datei mit Anweisungen zum Drucken eines Binär-Baumes mit Graphviz.

2.2 Herangehensweise

Die Datenstruktur AVL-Baum baut auf die simplere ADT Baum auf. In der vorliegenden Ausarbeitung gehe ich daher besonders auf die Besonderheiten des AVL-Baumes ein und lasse die Details zu Binär-Bäumen, soweit möglich, weg. Dieses Wissen über Binär-Bäume sei dem Leser dieser Ausarbeitung vorausgesetzt.

2.3 TDD in Erlang

Um mögliches Refactoring, bzw. schnelles Testen der Implementierung zu gewährleisten wurde vor den Implementierungen ein EUnit-Test geschrieben (siehe Datei *avltree_tests.erl*). Für einen Blackbox-Test, sprich der Test der exportierten Schnittstellen, kann nun diese Datei kompiliert werden und mit EUnit ausgeführt werden.

```
c(avltree_tests).
c(avltree).
eunit:test(avltree).
```

Für die Hilfsfunktionen wurden innerhalb des zu implementierenden Moduls (*avltree.erl*) jeweils Test-Methoden hinzugefügt, welche dann auch beim EUnit Test-Aufruf durchlaufen.

3 Entwurf und Implementierung der Lösung

Der abstrakte Datentyp AVL-Baum wird in Erlang mithilfe von Tupeln umgesetzt. Tupeln haben in Erlang eine feste Struktur und erlauben, im Vergleich zu Listen, weniger Operationen. Dadurch war schon von vornherein ein klarer Pfad notwendig, dem man bei der Implementierung befolgt. Zunächst wird festgelegt, dass das leere Tupel {} den leeren Baum repräsentiert und somit für die Initialisierung (initBT) des AVL-Baumes als Rückgabewert verwendet wird.
Für den AVL-Baum wird nun folgende rekursive Struktur für einen Knoten bzw. den gesamten Baum festgelegt:

```
B = {Element, Hoehe, LinkerTeilbaum, RechterTeilbaum}.
```

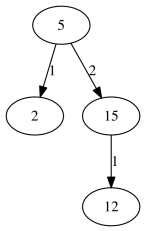
Ein Blatt sieht demnach so aus:

```
Blatt = {Element, Hoehe, {}, {}}
```

An erster Stelle ist das eigentliche Element des Knotens, welches gemäß der Aufgabenstellung nur ganze Zahlen erlaubt. Die zweite Stelle ist eine positive, ganze Zahl, beginnend ab 1 für einen nicht-leeren Baum, die die Höhe des Knotens im gesamten Baum speichert. Die eigentliche Rekursion beginnt nun an 3. und 4. Stelle in dieser Datenstruktur, welche jeweils wieder einen Knoten/Teil-Baum speichern. Ein Blatt ist ein Knoten, mit der Höhe 1 und leeren Tupeln für den linken und rechten Teilbaum, wie im folgenden Beispiel veranschaulicht wird:

```
B = { 5, 3,
      { 2, 1, {}, {} },
      { 15, 2,
        { 12, 1, {}, {} },
        {}
      }
    }.
```

Welches den folgenden Baum darstellt:



Mit dieser Struktur und dem Pattern-Matching in Erlang ist es nun trivial, den Baum rekursiv zu durchlaufen. Die Struktur für den Baum steht nun fest, die Operationen, wie Einfügen und Löschen aus dem Baum werden im Nachfolgenden erläutert. Dabei liegt das Augenmerk immer in der Erhaltung der AVL-Bedingung.

3.1 AVL-Bedingung

Damit ein Baum ein AVL-Baum ist, muss die AVL-Bedingung gelten. Dazu betrachtet man den **Balance-Faktor**, welcher definiert wird mit
$BF(t) = h(t_r) - h(t_l)$, wobei t_l und t_r jeweils den rechten und den linken Teilbaum von t darstellen und $h(t)$ die Höhe eines Teilbaumes ist. Ein Binär-Baum ist nun genau dann ein AVL-Baum wenn für alle Knoten in t gilt: $BF(t) \in \{-1, 0, 1\}$. Ein Baum mit dem Balance-Faktor 0 nennt man ausgewogen. Mit $BF(t) < 0$ ist ein Baum links- und mit $BF(t) > 0$ rechtslastig.

Für die Berechnung des Balance-Faktors in Erlang wird eine simple Funktion geschrieben:

```
balanceFaktor({}) -> 0;
balanceFaktor({_, _, {}, {}}) -> 0;
balanceFaktor({_, _, {_, HL, _, _}, {}}) -> 0 - HL;
balanceFaktor({_, _, {}, {_, HR, _, _}}) -> HR - 0;
balanceFaktor({_, _, {_, HL, _, _}, {_, HR, _, _}}) ->
  HR - HL.
```

Dank dem Pattern-Matching mit Tupeln in Erlang kann man so deklarativ den Balance-Faktor berechnen.
Diese Funktion wird nun benutzt, um eine eventuell benötigte Rotation zu erkennen und die richtige Rotation anzuwenden.

3.2 Rotationen

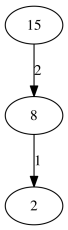
Beim Einfügen und beim Löschen in einen AVL-Baum muss vor und nach der Operation die AVL-Bedingung stets gelten. Um dies zu gewährleisten gibt es vier Wege eine Rotation an einem Knoten durchzuführen. Die *Links-Rotation*, *Rechts-Rotation*, *Doppel-Links-* und *Doppel-Rechts-Rotation*. Die Doppel-Links-Rotation kann man auch als *Rechts-Links-* und die Doppel-Rechts-Rotation als *Links-Rechts-Rotation* bezeichnen. Welche Rotation angewendet werden muss, kommt nun auf den Balance-Faktor eines Baumes an. Dazu wird eine Funktion geschrieben, die dies für einen Knoten feststellt (verkürzt):

```
1  checkAndRebalance(Baum) ->
2    BF_Ober = balanceFaktor(Baum),
3    if
4      BF_Ober == -2 ->
5            {_, _, TL, _} = Baum,
6            BF_Unter = balanceFaktor(TL),
7            if
8              BF_Unter == -1 -> rechtsRotation(Baum);
9              BF_Unter == 1 -> doppeltRechtsRotation(Baum)
10           end;
11         BF_Ober == 2 ->
12           % ... gleich, nur fuer Links-Rotation
13   end.
```

Erläuterung: In Zeile 2 wird zunächst der Balance-Faktor des Vater-Knotens berechnet (BF_Ober). Falls dieser -2 beträgt, ist der gegebene Baum linkslastig und es muss eine (Doppel-)Rechts-Rotation angewendet werden. Um nun zu entscheiden, welche Rotation angewandt wird wird der Balance-Faktor des linken Teilbaumes bestimmt (Zeile 4 und Zeile 8-9). Das gleiche wird für den Fall durchgeführt, falls BF_Ober gleich 2 ist. In diesem Fall wird dann natürlich eine (Doppel-)Links-Rotation angewandt.

3.2.1 Links- und Rechts-Rotation

Für die Rechts-Rotation wird der folgende Fall betrachtet:



Der Balance-Faktor für den Baum beträgt hier $0 - 2 = -2$. D.h. der Baum ist links-lastig. Der linke Teilbaum hat einen Balance-Faktor von $0-1 = -1$. Die *checkAndRebalance* Methode führt für diesen Knoten nun eine einfache Rechts-Rotation aus.

```
rechtsRotation({ E, _, L, R }) ->
  { LE, _, LL, LR } = L,
  NewRightNode = { E, berechneHoehe(LR, R), LR, R },
  NewNode = { LE,  berechneHoehe(LL, NewRightNode), LL, NewRightNode },
  NewNode.
```

Der neue Rechte Knoten wird nun das Element E, mit der Höhe die sich aus dem rechten Teilbaum des linken Teilbaumes und dem rechten Teilbaum des Ausgangs-Baumes zusammensetzt. Hierzu wird auch die Hilfsfunktion *berechneHoehe(L, R)* verwendet, die einfach nur $max(h(L), h(R)) + 1$ zurückgibt, wobei $h(B)$ die Höhe eines Baumes B ist. Der neue Knoten setzt sich nun zusammen aus dem Element des linken Teilbaumes, dem linken Teilbaum des linken Teilbaumes, und dem zuvor erwähnten neuen rechten Knoten.

So führt nun die Eingabe für den oberen Baum mit dem Tupel

```
B = {15, 3, {8, 2, {2, 1, {}, {}}, {}}, {}}
```

Zu der Ausgabe B1 mit

```
B1 = {8, 2, {2, 1, {}, {}}, {15, 1, {}, {}}}
```

Der Baum ist nach dieser Operation wieder balanciert. Auf die Links-Rotation wird jetzt nicht speziell eingegangen, da diese analog zur Rechts-Rotation auf dem rechten Teilbaum operiert.

3.2.2 Doppel-Rotationen

Eine Doppel-Rechts- bzw. Links-Rotation ist eine Zusammensetzung aus einfachen Links- und Rechts-Rotationen, die man geschickt anwendet. So wird bei einem gegebenen Baum B und seinen Teilbäumen B_L und B_R die Doppel-Rechts-Rotation wie folgt ausgeführt: Zunächst eine Links-Rotation auf B_L und dann eine Rechts-Rotation auf den gesamten Baum B. Analog dazu wird bei einer Doppel-Links-Rotation zunächst eine Rechts-Rotation auf B_R und dann eine Links-Rotation auf auf den gesamten Baum B angewandt. Im Code sieht das für die Doppel-Rechts-Rotation dann folgendermaßen aus:

```
doppeltRechtsRotation({E, H, L, R}) ->
  rechtsRotation({ E, H, linksRotation(L), R }).
```

3.3 Prüfen, ob ein AVL-Baum vorliegt

Zur Überprüfung eines AVL-Baumes wird die rekursive Funktion $isBT(B)$ verwendet, welche *true* für einen gültigen AVL-Baum liefert, ansonsten $false$.

```
1  isBT(B) ->
2    {E, H, L, R} = B,
3    {LE, LH, _, _} = L,
4    {RE, RH, _, _} = R,
5    ValueCorrect = middle(E, LE, RE),
6    HeightCorrect = (max(LH, RH) + 1) == H,
7    BalanceIsValid = (abs(balanceFaktor(B)) =< 1),
8    BalanceIsValid and
9    HeightCorrect and
10   ValueCorrect and
11   isBT(L) and isBT(R).
```

Erläuterung:
Zeile 5: Es wird überprüft, ob der Wert E des aktuellen Baumes B zwischen dem Wert des linken und dem Wert des rechten Teilbaumes von B liegt. Hierzu wird die Hilfsfunktion *middle* benutzt.
Zeile 6: Es wird überprüft, ob die Höhe des aktuellen Baumes B korrekt ist. Die Höhe ist dann korrekt, wenn $max(h(B_L), h(B_R)) + 1 == h(B)$.
Zeile 7: Es wird nun mithilfe der zuvor definierten Hilfs-Funktionen geschaut ob der Balance-Faktor des aktuellen Baumes B im gültigen Bereich von -1 bis 1 liegt.
Zeile 11: Der rekursive Aufruf für jeweils den linken und den rechten Teilbaum wird gestartet.
Ferner, und hier nicht zu sehen, gibt die Funktion für einen leeren Baum B *true* zurück und beendet die Rekursion spätestens dort.

3.4 Einfügen in einen AVL-Baum

Das rekursive Einfügen in einen AVL-Baum muss dafür sorgen, dass zusätzlich zum Einfügen wie in einen Binär-Baum die AVL-Bedingung erhalten bleibt.

`insertBT(B, N) -> B.`

Objektmengen: N = Einzufügendes Element, E = Element des aktuellen Knotens, B = (Teil-)Baum

1. Prüfe, ob einzufügendes Element N vom Typ *Integer* ist. Falls ja, mache weiter. Falls nein, kann hier abgebrochen werden.
2. Falls $N > E$, wird das Element rekursiv in den rechten Teilbaum B_R von B eingefügt: $insertBT(B_R, N)$
3. Prüfe, ob sich die Höhe verändert hat nach der vorherigen Einfüge-Operation. Falls ja, prüfe an dem aktuellen Knoten die AVL-Bedingung und führe evtl. eine Rotation durch. Falls nein, und die Höhe hat sich nicht geändert ist auch keine Rebalancierung notwendig.
4. (Analog für $N < E$)

Die Rekursion wird aufgelöst, sobald das Element an einen Teil-Baum als Blatt hinzugefügt worden ist. Hierzu wird die Funktion $insertBT(\{\}, N)$ definiert, welche nur einen neuen Baum mit N als Element, der Höhe 1 und jeweils leerem linken und rechten Teilbäumen zurückgibt.

3.5 Löschen aus einem AVL-Baum

Das Löschen eines Knotens aus einem AVL-Baum funktioniert grob nach folgendem Algorithmus:

Objektmengen: D = Zu löschendes Element, B = Baum

1. Gehe zum zu löschenden Knoten durch Traversion von B
2. Knoten wurde gefunden: Ersetze den gefundenen Knoten mit dem größten Element aus dem linken Teilbaum bzw., falls linker Teilbaum leer, mit dem kleinsten Element aus dem rechten Teilbaum und lösche diesen (rekursiv) Knoten
3. Falls der zu löschende Knoten ein Blatt ist, lösche es

Der Quell-Code dazu sieht so aus:

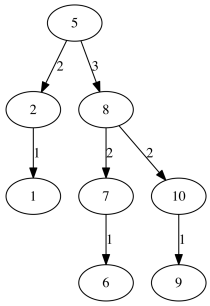
```
1  deleteBT({E, H, L, R}, D) ->
2    LeftEmpty = isEmptyBT(L),
3    RightEmpty = isEmptyBT(R),
4    if
5      D > E ->
6        NewRight = deleteBT(R, D),
7        NewHeight = berechneHoehe(L, NewRight),
8        checkAndRebalance({ E, NewHeight, L, NewRight });
9      D < E ->
10         % ... gleiches fuer linken Teilbaum;
11     D == E ->
12       if
13         H == 1 -> {};
14         not LeftEmpty ->
15           BiggestLeftChild = findBiggest(L),
16           NewLeft = deleteBT(L, BiggestLeftChild),
17           NewHeight = berechneHoehe(NewLeft, R),
18           checkAndRebalance({BiggestLeftChild, NewHeight, NewLeft, R});
19         not RightEmpty ->
20           SmallestRightChild = findSmallest(R),
21           NewRight = deleteBT(R, SmallestRightChild),
22           NewHeight = berechneHoehe(L, NewRight),
23           checkAndRebalance({ SmallestRightChild, NewHeight, L, NewRight })
24       end;
25     true -> {E, H, L, R }
26   end.
```

Erläuterung: In Zeile 2-3 prüfen wir zunächst, ob der linke und rechte Teilbaum leer ist, mit der bereits vorhandenen Methode $isEmptyBT(B)$. Ab Zeile

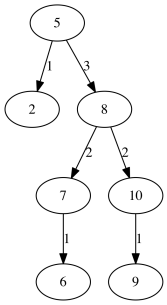
4 werden insgesamt 3 Fälle geprüft: Ist der zu löschende Knoten größer als der aktuelle Knoten, so wird die Suche im rechten Teilbaum durch einen rekursiven Aufruf in Zeile 6 fortgesetzt. Das Ergebnis hierraus wird der Variable $NewRight$ zugewiesen. In den folgenden Zeilen 7-8 werden nun die Höhe angepasst und mit einem Aufruf an *checkAndRebalance* eventuelle Rotationen durchgeführt. Zeile 11 behandelt den Fall, dass der gesuchte Knoten nun der aktuelle Knoten ist. Falls dieser Knoten ein Blatt ist (zu erkennen an der Höhe 1) wird einfach der leere Baum zurückgegeben. Falls nicht, wird im linken oder rechten Teilbaum nach dem größten bzw. kleinsten Knoten gesucht und oben erwähnter Algorithmus wird angewendet. Zuguterletzt werden nun noch die Höhe angepasst und per Aufruf an *checkAndRebalance* eventuelle Rotationen durchgeführt.

3.5.1 Sonderfall für *checkAndRebalance*

Betrachten wir für das Löschen folgenden Fall:



Aus diesem Baum wird der Knoten 1 aus dem linken Teilbaum gelöscht. Der Baum sieht dann (vor Rebalancierung) so aus:



Die Methode *checkAndRebalance* würde hier für den Balance-Faktor des Vater-Knotens 2 bestimmen und für den Kind-Knoten eine Faktor von 0. Dieser Fall war für $insertBT$ belanglos, da er bei sukzessiver Rebalancierung beim Einfügen nie auftreten konnte. Für $deleteBT$ muss dieser Fall aber behandelt werden und wird somit in *checkAndRebalance* auch aufgeführt.

3.6 Ausgabe als .dot-Datei, *printBT*

Die hier gewählte Methode der Darstellung eines Baumes ist zwar gut lesbar und manipulierbar für eine Maschine, jedoch ab einer bestimmten Größe nicht mehr nachvollziehbar für einen normalen, menschlichen Betrachter. So wird durch die Erstellung einer .dot-Datei und dem Tool GraphViz[1] die Visualisierung eines Baumes ermöglicht. Hierzu wird die Funktion $printBT(B, Filename)$ wie folgt implementiert (verkürzt dargestellt):

```
1  printBT(BTree, Filename) ->
2    writeHead(Filename),
3    startPrint(Filename, BTree),
4    writeFoot(Filename).
5
6  startPrint(Filename, {X, _, Left, Right}) ->
7    { LeftNodeValue, LeftNodeHeight, _, _ } = Left,
8    { RightNodeValue, RightNodeHeight, _, _ } = Right,
9    writeLine(Filename, {X, LeftNodeValue}, LeftNodeHeight),
10   writeLine(Filename, {X, RightNodeValue}, RightNodeHeight),
11   startPrint(Filename, Left),
12   startPrint(Filename, Right).
```

Erläuterung: Die Methode *printBT* nimmt einen Binär-Baum und startet u.a. den Aufruf an die rekursive Methode *startPrint*. Diese Schreibt nun Zeile für Zeile die Knoten des Binär-Baums in die .dot-Datei und wiederholt den Schritt für jeweils den linken und rechten Teilbaum.
Eine Beispiel-Ausgabe sieht folgendermaßen aus:

```
digraph avltree
{
5 -> 2 [label = 3];
5 -> 8 [label = 3];
2 -> 1 [label = 1];
}
```

Aus dieser Ausgabe kann nun mit dem Befehl *dot -Tsvg test.dot > test.svg* in der Kommando-Zeile die Graphik erzeugt werden.

[1] siehe http://graphviz.org/

3.7 Laufzeitmessungen

In der Theorie wird für den AVL-Baum für Einfüge-, Lösch- und Such-Operationen eine Laufzeit von $O(log * n)$ angegeben. Dies wird anhand dieser Implementierung getestet. Dazu wird eine Test-Umgebung errichtet, welche eine Liste der Größe n mit zufällig gewählten Elementen in einen AVL-Baum einfügt[2]. Es wurden im Mittel die folgenden Zeiten gemessen:

[2] siehe Datei experimente.erl

4 Quellen