5.3 AVL-Bäume

In diesem Abschnitt werden wir AVL-Bäume betrachten, die auf Adelson-Velskij und Landis zurückgehen. Mit diesen ist es möglich das Wörterbuchproblem so zu lösen, dass alle drei Operationen, selbst im schlechtesten Fall, eine logarithmische Laufzeit haben. Dazu benötigen wir noch einige vorbereitende Begriffe und Beobachtungen.

Die *Höhe* $\mathrm{h}(v)$ eines Knotens v ist die maximale Anzahl von Knoten auf einem Pfad von v zu einem Blatt. Die Höhe des leeren Baums ist 0. Die Höhe eines nicht-leeren Baums ist die Höhe seiner Wurzel und damit ≥ 1.

Definition 5.2. Sei T ein Suchbaum, v ein Knoten in T, T_l der linke Teilbaum von v und T_r der rechte Teilbaum von v. Dann ist der *Balancegrad* von v definiert als $\mathrm{bal}(v) = \mathrm{h}(T_\mathrm{r}) - \mathrm{h}(T_\mathrm{l})$. Ein Suchbaum T heißt *balanciert* falls für jeden Knoten v von T gilt: $|\mathrm{bal}(v)| \leq 1$.

Satz 5.2. *Ein balancierter Baum mit n Knoten hat Höhe $\Theta(\log n)$.*

Beweis. Ein balancierter Baum mit Höhe h und maximaler Anzahl von Knoten ist der vollständige Binärbaum. Dieser hat $n = \Theta(2^h)$ Knoten. Für einen beliebigen balancierten Baum ist also $h = \Omega(\log n)$.

Ein balancierter Baum mit Höhe h und minimaler Anzahl von Knoten hat für $h = 1$ einen Knoten und für $h = 2$ zwei Knoten. Für $h \geq 3$ hat er zwei Teilbäume, einen davon ein balancierter Baum mit Höhe $h - 1$ und minimaler Anzahl von Knoten, den anderen ein balancierter Baum mit Höhe $h - 2$ und minimaler Anzahl von Knoten. Wir erhalten also für die minimale Anzahl von Knoten n_h in einem balancierten Baum mit Höhe h die Rekursionsgleichung

$$n_h = n_{h-1} + n_{h-2} + 1 \text{ mit } n_1 = 1 \text{ und } n_2 = 2.$$

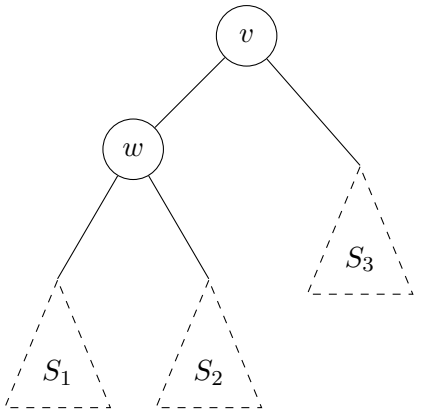
Man sieht sofort dass $n_h \geq F_h$. Wir wissen bereits dass

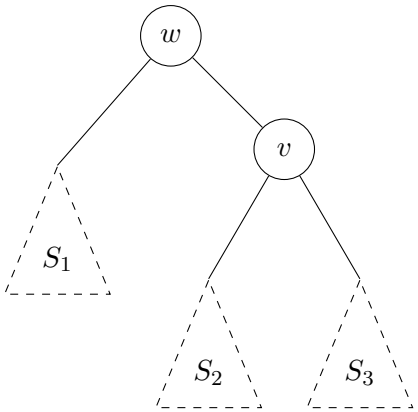
$$F_h = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^h - \left(\frac{1-\sqrt{5}}{2} \right)^h \right)$$

Nun geht $\left(\frac{1-\sqrt{5}}{2}\right)^h \to 0$ für $h \to \infty$ und damit $F_h = \Theta(\Phi^h)$ für den goldenen Schnitt $\Phi = \frac{1+\sqrt{5}}{2}$. Also $n_h = \Omega(\Phi^h)$. Für einen beliebigen balancierten Baum gilt also $h = O(\log_\Phi n) = O(\log n)$. Insgesamt gilt also für einen beliebigen balancierten Baum $h = \Theta(\log n)$. □

Der wesentliche Punkt an einem AVL-Baum wird nun sein, dass er ein balancierter Suchbaum ist. Um die Balanciertheit beizubehalten müssen die Einfüge- und Lösch-Operationen modifiziert werden. Das wird durch Korrekturtransformationen erreicht, die sich auf die folgenden Rotationsoperationen stützen.

Definition 5.3. Seien

$T_1 =$ 

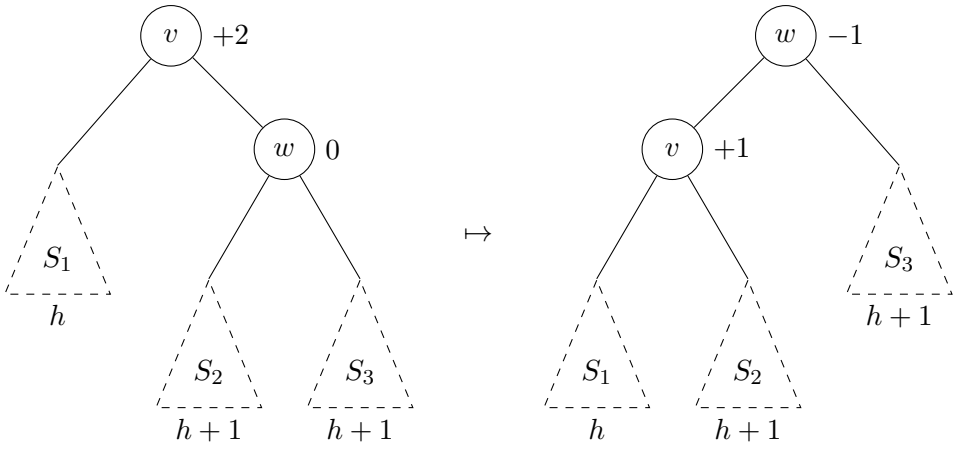
und $T_2 =$ 

wobei v und w Knoten sind und S_1, S_2 und S_3 Bäume. Dann bezeichnen wir die Abbildung von T_1 auf T_2 als Rechtsrotation (an der Stelle v) und die Abbildung von T_2 nach T_1 als Linksrotation (an der Stelle w).

Die Suchbaumeigenschaft wird durch diese Rotationen beibehalten. Seien nämlich x_1, x_2 und x_3 Schlüssel in S_1, S_2 bzw. S_3, dann gilt $x_1 < w.D.x < x_2 < v.D.x < x_3$ sowohl in T_1 als auch in T_2.

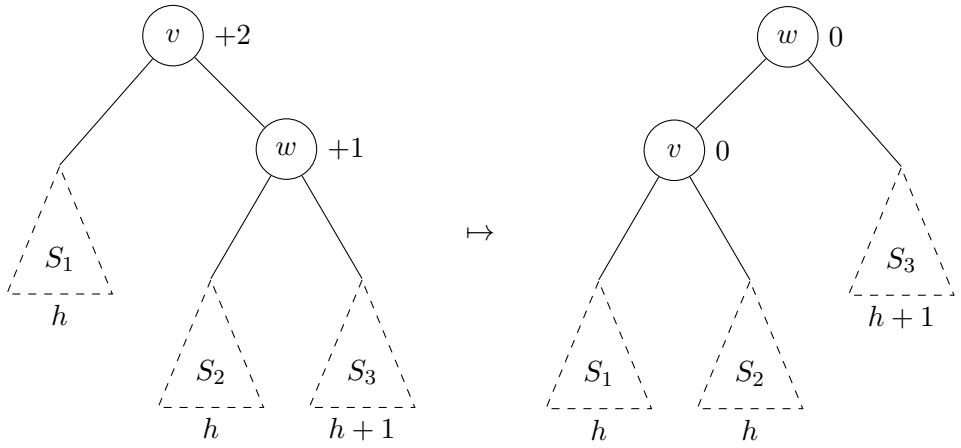
Konkret entwickeln wir die Korrekturtransformationen wie folgt: Zunächst einmal stellen wir fest, dass die Eigenschaft der Balanciertheit unabhängig von den Schlüsseln der Datensätze ist, sondern nur von der Form des Baums abhängt. Die Einfügeoperation verändert die Form des Baums indem sie ein neues Blatt hinzufügt, die Löschoperation indem sie ein bestehendes Blatt entfernt. Der Balancegrad aller Knoten die nicht auf dem Pfad von der Wurzel zur Stelle der Änderung liegen bleibt also unverändert. Der Balancegrad der Knoten auf diesem Pfad verändert sich um höchstens 1. Insgesamt haben wir bei der Korrektur der Form des Baums also mit $O(h)$ Knoten zu tun deren Balancegrad in $\{-2, -1, 0, 1, 2\}$ liegt. Wir skizzieren eine Prozedur Balancieren(v). Die Prozedur erhält einen Baum T über dessen Wurzel v als Eingabe. Von T wird angenommen, dass $|\text{bal}(v)| \leq 2$ und für alle Knoten $w \in T \setminus \{v\}$ gilt: $|\text{bal}(w)| \leq 1$. Die Prozedur gibt einen balancierten Suchbaum mit den Elementen von T zurück. Balancieren(v) geht wie folgt vor:

1. Falls $|\text{bal}(v)| \leq 1$, dann antworte mit v.
2. Falls $\text{bal}(v) = +2$, dann existiert ein Knoten $w = v.\textit{rechts}$.
   (a) Falls $\text{bal}(w) = 0$, dann führen wir die folgende Linksrotation durch



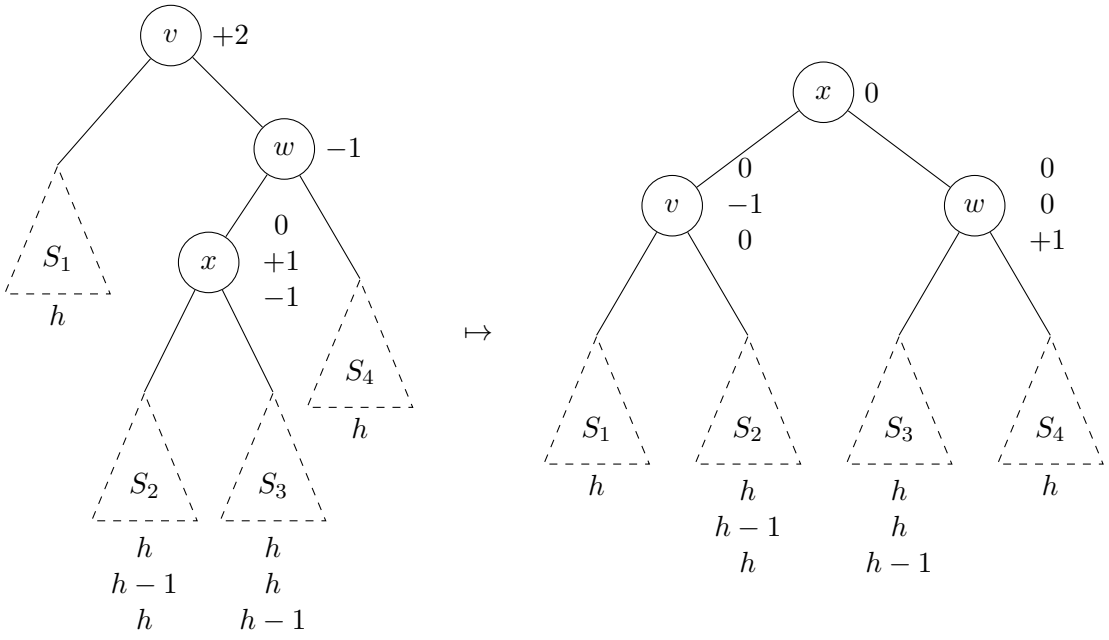
und antworten mit w. Der Eingabebaum hat Tiefe $h+3$. Der Ausgabebaum hat ebenfalls Tiefe $h+3$.

(b) Falls $\text{bal}(w) = +1$, dann führen wir die folgende Linksrotation durch



und antworten mit w. Der Eingabebaum hat Tiefe $h+3$. Der Ausgabebaum hat Tiefe $h+2$.

(c) Falls $\text{bal}(w) = -1$, dann existiert ein Knoten $x = w.\mathit{links}$. Wir führen die folgende Doppelrotation durch



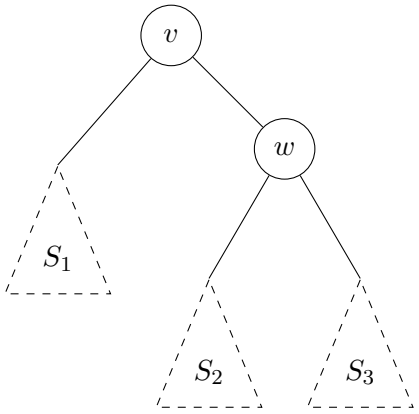
und antworten mit x. Der Eingabebaum hat Tiefe $h+3$. Der Ausgabebaum hat Tiefe $h+2$.

3. Der Fall $\text{bal}(v) = -2$ ist symmetrisch zum vorherigen.

Ein AVL-Baum erlaubt nun eine effiziente Implementierung dieser Prozedur indem jeder Knoten v, zusätzlich zu $v.D$, $v.\mathit{links}$, $v.\mathit{rechts}$ noch ein Feld $v.\mathit{bal}$ für den Balancegrad von v hat. Der Balancegrad wird im Knoten gespeichert um die (linearen) Kosten zu seiner Bestimmung einzusparen. Damit muss der in den Knoten gespeicherte Balancegrad natürlich auch durch alle

ändernden Operationen aktualisiert werden. Erreicht ist dadurch dass die Prozedur BALANCIEREN eine Laufzeitkomplexität von $\Theta(1)$ hat.

Die Operationen in AVL-Bäumen können dann folgendermaßen implementiert werden. Die Suchoperation in einem AVL-Baum ist wie die in einem allgemeinen Suchbaum. Die Einfüge- und Lösch-operationen müssen an jeder Stelle v den neues Balancegrad $\text{bal}(v)$ berechnen, und, zumindest falls $|\text{bal}(v)| = 2$, die Prozedur BALANCIEREN(v) aufrufen. Den neuen Balancegrad auf die naive Weise zu berechnen, d.h. durch Bestimmung der Höhen der Teilbäume, ist keine Option, da dies lineare Zeit benötigen würde. Um den neuen Balancegrad in konstanter Zeit zu berechnen, muss er aus dem alten Balancegrad mit Hilfe von Informationen berechnet werden, die den Einfüge- und Löschoperationen zur Verfügung stehen. Konkret wird dazu die Veränderung der Höhe der Teilbäume benötigt. Diese beiden Operationen müssen also im rekursiven Aufstieg die Information mitführen, ob der aktuelle Teilbaum seine Höhe verändert hat, d.h. ob sie um 1 gestiegen ist (im Fall von Einfügen) oder um 1 gefallen (im Fall von Löschen). Zur Illustration geben wir hier ein Beispiel. Wir betrachten einen Baum der Form



in dem EINFÜGEN(v, D) den Aufruf EINFÜGEN(w, D) gemacht hat (d.h. der Schlüssel x des einzufügenden Elements war größer als $v.D.x$). Wir gehen davon aus, dass die Prozedur EINFÜGEN so modifiziert wurde, dass sie mit einem Paar $(v, \Delta h)$ antwortet wobei v, wie gehabt, der neue Baum ist und Δh die Differenz zwischen der Höhe des neuen Baums und der Höhe des alten Baums. Ebenso wird die Prozedur BALANCIEREN so modifiziert, dass sie mit dem Paar $(v, \Delta h)$ antwortet wobei v wiederum der neue Baum ist und Δh die Höhenänderung vom alten zum neuen Baum. Sei nun $(w, \Delta h_w)$ die Antwort von EINFÜGEN(w, D). Der Balancegrad von v sowie Δh_v berechnen sich dann wie folgt:

$v.bal := v.bal + \Delta h_w$
$(v, \Delta_b) :=$ BALANCIEREN(v)
$\Delta h_v := \Delta h_w + \Delta_b$

An den anderen Stellen des Pseudocodes von Einfügen und Löschen werden ähnliche Änderungen durchgeführt. Diese Operationen haben damit eine Laufzeit von $O(\log n)$ wobei n die Anzahl der Datensätze ist.

Logarithmische Komplexität ist in der Praxis sehr gering. Zum Beispiel kann ein Suchbaum der alle ca. 8 Mrd. Menschen die auf der Erde leben enthält mit einer Tiefe von ca. 33 erstellt werden. Mit AVL-Bäumen haben wir also das Wörterbuchproblem auf zufriedenstellende Weise gelöst.

Suchbäume erlauben auch das folgende Sortierverfahren: ein gegebenes Datenfeld A kann sortiert werden durch 1. einen Aufbau eines Suchbaums für A durch wiederholten Einfügen in den leeren Baum sowie 2. den Durchlauf dieses Suchbaums in symmetrischer Reihenfolge, siehe

Algorithmus 13. Der zweite Schritt benötigt Laufzeit $\Theta(n)$. Bei Verwendung von AVL-Bäumen benötigt der erste Schritt Zeit $O(n \log n)$ und der gesamt Algorithmus damit ebenfalls $O(n \log n)$.

5.4 Stapel und Warteschlangen

Ein Stapel (engl. *stack*) ist eine Datenstruktur welche die folgenden Operationen zur Verfügung stellt:

1. *Hinzufügen*(S, D) legt den Datensatz D auf den Stapel S (engl. *push*).
2. *Entfernen*(S) gibt das oberste Element vom Stapel zurück und entfernt es vom Stapel (engl. *pop*).

Dieses Prinzip bezeichnet man auch als LIFO "last in first out". Ein Stapel kann als einfach verkettete Liste S mit einem Startzeiger $S.Start$ implementiert werden, wobei der Stapel leer ist genau dann wenn $S.Start = \textsc{Nil}$, siehe Algorithmus 18. Diese Operationen benötigen jeweils Zeit $\Theta(1)$.

Algorithmus 18 Stapel

```
Prozedur HINZUFÜGEN(S, D)
    Sei v neuer Knoten
    v.D := D
    v.nächster := S.Start
    S.Start := v
Ende Prozedur

Prozedur ENTFERNEN(S)
    Falls S.Start = NIL dann
        Antworte "Stapel leer"
    sonst
        D := S.Start.D
        S.Start := S.Start.nächster
        Antworte D
    Ende Falls
Ende Prozedur
```

Wir wollen nun eine Warteschlange (ähnlich wie an einer Supermarktkassa) implementieren. Es soll möglich sein, das erste Element aus einer Warteschlange zu entfernen (z.B. um es abzuarbeiten) sowie ein neues Element an das Ende der Warteschlange zu stellen. Die Elemente werden also in der Reihenfolge ihres Einlangen abgearbeitet. Dieses Prinzip bezeichnet man auch als FIFO "first in first out", die gewünschte Datenstruktur als Warteschlange (engl. *FIFO queue*). Wir wollen die folgenden Operationen

1. *Hinzufügen*(W, D) stellt D an das Ende der Warteschlange W.
2. *Entfernen*(W) gibt das Element vom Anfang der Warteschlange zurück und entfernt es.

Eine Möglichkeit eine solche Warteschlange W zu implementieren besteht darin eine einfach verkettete Liste zu verwenden mit dem Startzeiger $W.Start$ und einem zusätzlichen Endezeiger $W.Ende$. Die Implementierung in Algorithmus 19 geht davon aus, dass die Warteschlange

Algorithmus 19 Warteschlange

```
Prozedur HINZUFÜGEN(W, D)
    Sei v neuer Knoten
    v.D := D
    v.nächster := NIL
    Falls W.Start = NIL dann
        W.Start := v
    sonst
        W.Ende.nächster := v
    Ende Falls
    W.Ende := v
Ende Prozedur

Prozedur ENTFERNEN(W)
    Falls W.Start = NIL dann
        Antworte "Warteschlange leer"
    sonst
        D := W.Start.D
        W.Start := W.Start.nächster
        Antworte D
    Ende Falls
Ende Prozedur
```

leer ist genau dann wenn der Startzeiger NIL ist. Wie man leicht sehen kann benötigen diese Operationen jeweils $\Theta(1)$ Zeit.