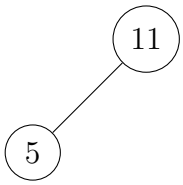
Aufgabe 1

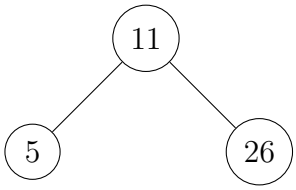
(a)

Füge 11 hinzu

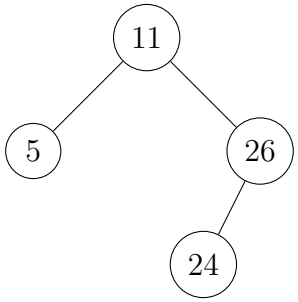
Füge 5 hinzu



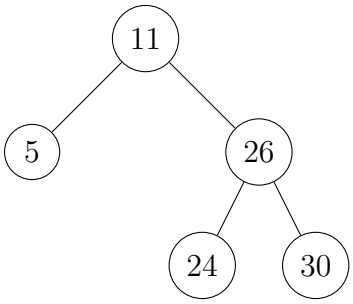
Füge 26 hinzu



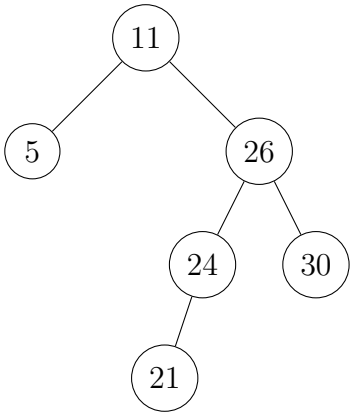
Füge 24 hinzu



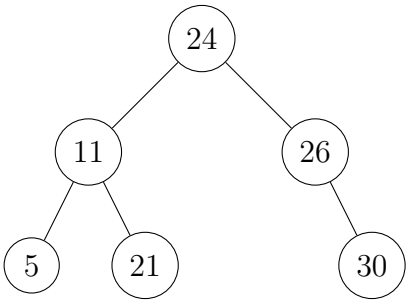
Füge 30 hinzu



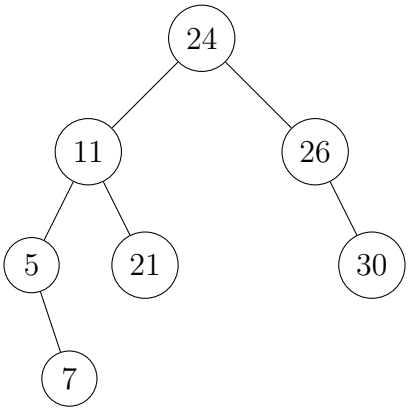
Füge 21 hinzu



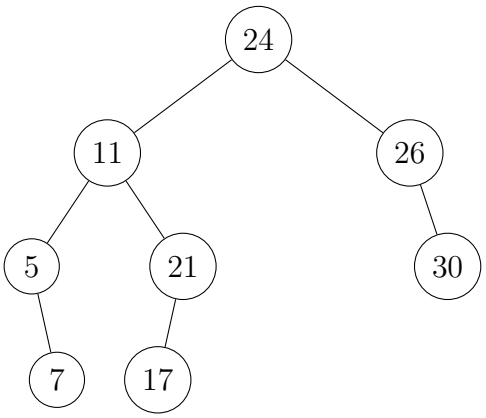
Doppelrotation



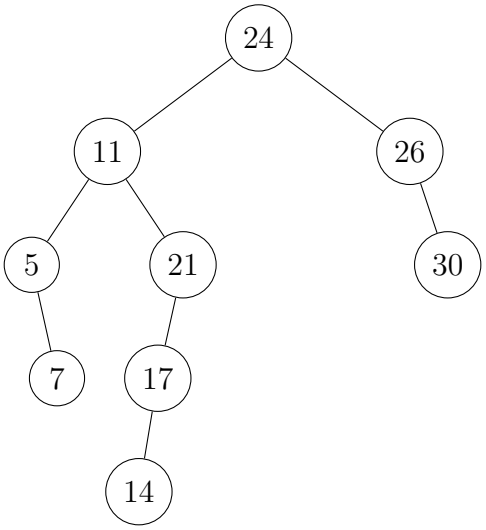
Füge 7 hinzu



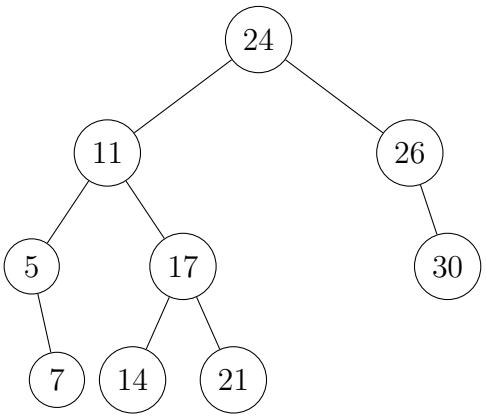
Füge 17 hinzu



Füge 14 hinzu

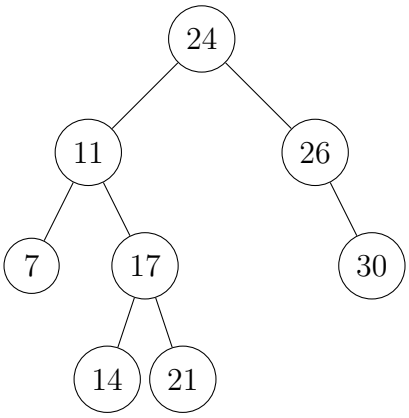


Rotation

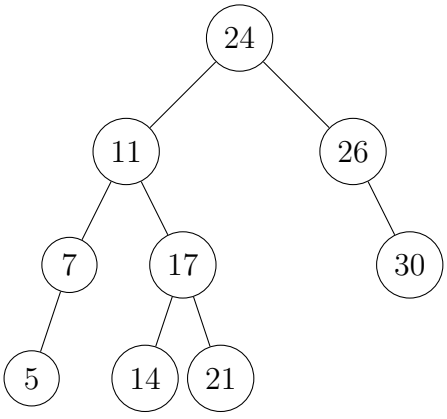


(b)

Entferne 5



Füge 5 wieder hinzu

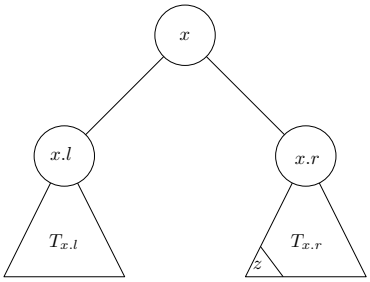


Aufgabe 2

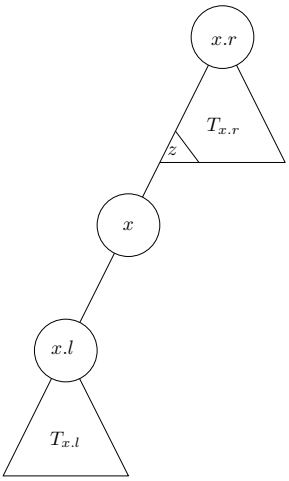
(a) Sei x das zu löschende Element und T_x der Baum mit x als Wurzel und $T_{x.l}$ und $T_{x.r}$ der Teilbaum des linken und rechten Kindes von x. Wir wissen, dass $x.r \neq \mathsf{TNULL}$ gilt, da wir sonst in einem anderen Fall wären. Somit enthält $T_{x.r}$ mindestens ein Element. Sei z das kleinste Element in $T_{x.r}$.

Der angegebene Algorithmus führt folgende Umwandlung auf dem Baum durch:

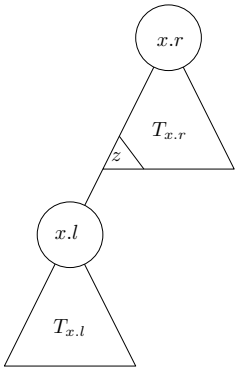
Ursprünglicher Baum:



Baum nach dem Rotieren:



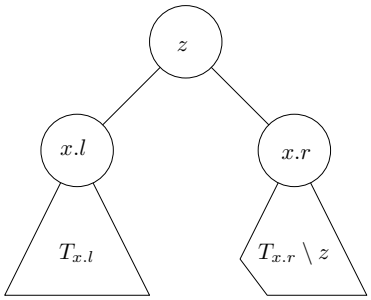
Baum nach dem Löschen:



Formal kann die Korrektheit dieser Umwandlung noch per Induktion gezeigt werden.

(b) Es ist einfach zu sehen, dass wir im Grunde für (fast) jede der Links-Rotationen wieder eine Rechts-Rotation durchführen müssen. Nur die letzte Linksrotation, die dazu geführt hat, dass x kein rechtes Kind mehr hat, braucht nicht „rückgängig“ gemacht zu werden.

Dies erzeugt folgenden Baum:



Nach diesen Rotationen muss „nur noch“ die möglicherweise verletzte AVL-Eigenschaft in $T_{x.r} \setminus z$ wiederhergestellt werden. Dies ist aber identisch mit dem Vorgang beim „klassischen“ Löschen und wird somit hier nicht betrachtet.

Aufgabe 3

(a) Eine solche Familie sind Fibonacci-Bäume[1]. Diese sind ähnlich wie die Fibonacci-Zahlen rekursiv definiert: F_0 ist der leere Baum und F_1 ein einzelnes Blatt. Der Baum F_n ist nun eine Wurzel mit F_{n-1} als linkem Kind und F_{n-2} als rechtem Kind. Die rechten Unterbäume haben also immer eine um 1 geringere Höhe als die linken Unterbäume, somit sind alle Teilbäume unbalanciert während sie gleichzeitig die AVL-Eigenschaft erfüllen.
Entfernt man nun den größten (rechtesten) Knoten, ist der Teilbaum, in dem dieser enthalten ist um 2 unbalanciert und es muss rotiert werden. Der durch die Rotation entstandene Teilbaum hat nun eine geringere Höhe, somit ist an seinem Elter eine Höhendifferenz von 2 und es muss wieder rotiert werden. Diese Höhenverringerung wird bis zur Wurzel hochpropagiert und für jeden Knoten auf dem Weg wird rotiert. Also werden Rotationen durchgeführt entsprechend der Tiefe des größten Knotens. Diese Tiefe entspricht exakt der Hälfte der Tiefe des gesamten Baumes. (Das lässt sich leicht dadurch erkennen, dass der größte (rechteste) Knoten sich in einem F_1 Baum befindet, der das rechte Kind eines F_3 Baumes ist und alle weiteren Vorfahren zur Wurzel immer dem zweitnächsten Fibonacci-Baum entsprechen, während das tiefste Blatt, der kleinste (linkeste) Knoten, sich in einem F_1 Baum befindet der das linke Kind eines F_2 Baumes ist und alle weiteren Vorfahren immer dem nächsten Fibonacci-Baum entsprechen.) Da es sich um einen AVL-Baum handelt ist die Höhe des tiefsten Knoten und damit auch die Höhe des rechtesten Knoten in $\Theta(\log n)$ Also ist die Anzahl der Rotationen logarithmisch in der Anzahl der Knoten.

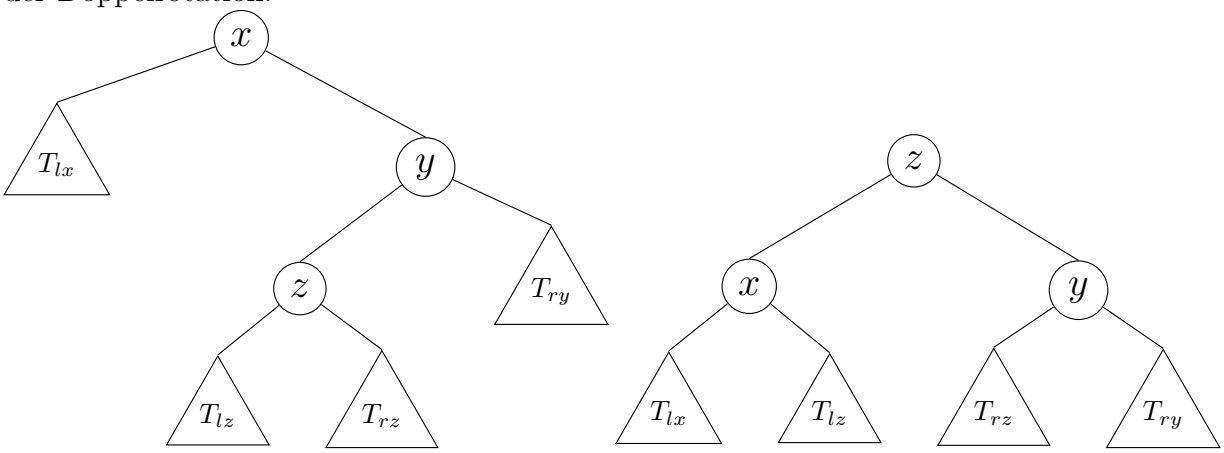
(b) Es kann höchstens eine Doppelrotation geben, also zwei Rotationen. Im nachfolgenden bezeichnen wir die Höhendifferenz der Unterbäume eines Knotens als Balance. Es werden im insert aus der Vorlesung auf dem Pfad vom eingefügten Knoten bis zur Wurzel alle Teilbäume auf die AVL-Eigenschaft überprüft. Zunächst einmal müssen wir nur die Fälle betrachten, in denen der eingefügte Knoten die Balance seines Elterknotens nicht auf Null setzt, da nur in diesem Fall die Höhe des AVL-Baumes verändert wird. Nehmen wir also an das Einfügen habe die Höhe erhöht und sich die Balance dementsprechend verändert. Wir unterscheiden zwei Fälle

- Die Balance hat sich auf -1 (1) verändert.
  Die AVL-Eigenschaft ist also noch erfüllt, da sich aber die Höhe unseres Baumes geändert hat, müssen wir noch die höheren Knoten auf dem Pfad zur Wurzel überprüfen.
  Wir haben aber noch keine Rotation durchgeführt.
- Die Balance hat sich auf -2 (2) verändert.
  Die AVL-Eigenschaft ist verletzt, wir betrachten nur -2 da der andere Fall gespiegelt und somit analog ist. Somit ist der rechte Unterbaum um zwei tiefer als der linke. Es muss in den rechten Unterbaum eingefügt worden sein, da sich sonst die AVL-Eigenschaft nicht verletzt worden sein könnte. Desweiteren muss der rechte Unterbaum eine Balance von 1 oder -1 haben, da sich sonst die Tiefe des Gesamtbaumes nicht verändert hätte (siehe oben). Wir unterscheiden wieder zwischen diesen beiden Fällen:

[1] `https://de.wikipedia.org/wiki/Fibonacci-Baum`

Falls die Balance -1 ist, führen wir eine Links-Rotation aus, da der rechte Teilbaum der tiefere ist. Nun ist die Baumtiefe von vor dem insert wiederhergestellt, daher ist keine weitere Rotation mehr nötig.
Falls die Balance 1 ist, führen wir eine Doppelrotation durch (siehe unten). Man beachte, dass die Bäume $h(T_{lx}) = h(T_{ry}) = max(h(T_{lz}), h(T_{rz}))$ gilt und daher die Tiefe nach der Rotation wieder der vor dem Einfügen entspricht.
Wir haben also zwei Rotationen durchgeführt und es sind keine weiteren mehr nötig.

Da wir je nach Fall keine Rotation durchführen und auf dem Pfad weiter nach oben gehen, oder aber Rotationen durchführen und anschließend garantiert einen AVL-Baum haben, werden maximal zwei Rotationen pro Einfügen durchgeführt. Beispiel der Doppelrotation:



Aufgabe 4A

Wie verwalten alle Elemente in einem AVL-Baum geordnet nach den Beträgen. Zusätzlich zu Höhe, dem Betrag als Schlüssel und den Zeigern auf die Kinder speichern wir in jedem Knoten die folgenden Felder:

```
class Knoten {
    Knoten left_child, right_child; // Kinder
    Knoten parent; // Elter
    int height; // die Hoehe das Unterbaums
    double unsigned_a; // der Schluessel |a|
    boolean positive;  // wahr wenn a > 0
    int size; // Anzahl der Knoten im Unterbaum
    int total_vzw;  // VZW fuer die Elemente im Unterbaum
    boolean min_positive; // 'positive' des kleinsten Nachfahren
    boolean max_positive; // 'positive' des groessten Nachfahren
}
```

Diese speziellen Werte lassen sich für einen Knoten in konstanter Zeit berechnen, wenn sie für die Kinder bekannt sind. (Z.B. lässt sich aus `max_positive` des linken Kindes und `min_positive` des rechten Kindes ableiten, ob ein Vorzeichenwechsel mit der Wurzel stattfindet, alle anderen VZW sind für die Kinder bereits bestimmt)

Es ist also möglich `insert` und `delete` wie in einem gewöhnlichen AVL-Baum zu implementieren und dabei die Korrektheit dieser Werte aufrechtzuerhalten, indem am Ende die Werte auf dem Pfad von der Änderung bis zur Wurzel neu berechnet werden.

Für die `vzw`-Anfrage betrachten wir die Pfade von a_l und a_r zu ihrem niedrigstem gemeinsamen Vorfahren. Nach der Suchbaum Eigenschaft befinden sich die Elemente aus $S[l, r]$ in den rechten Teilbäumen auf dem Pfad zu l und den linken Teilbäumen auf dem Pfad zu r. (Wenn sich a_l im linken Teilbaum eines Knotens befindet so ist $|a_l|$ kleiner als der Schlüssel und die Elemente im rechten Teilbaum haben einen Index größer l, gleichzeitig sind diese Teilbäume im linken Teilbaum eines Vorfahren von r und besitzen entsprechend einen Index kleiner r, entsprechendes gilt für die linken Teilbäume auf dem Pfad zu r). Die Vorzeichenwechsel lassen sich aus den gespeicherten `vzw`- und den `positive`-Werten dieser Teilbäume bestimmen.

Da uns die Werte $|a_l|$ und $|a_r|$ nicht bekannt sind suchen wir mit Hilfe des Indexes l, r. Dazu benötigen wir die `size`-Werte: In einem Baum mit kleinstem Index i_0 und einem linken Teilbaum der Größe n besitzt die Wurzel den Index $n + i_0$ und die Indices i_0 bis $n - 1 + i_0$ befinden sich im linken Teilbaum während sich alle Indices größer $n + 1 + i_0$ im rechten Teilbaum befinden.

```
int vzw(int l, int r){ // in Knoten this
    int res = 0;
    Knoten lk = this.search_index(l);
    Knoten rk = this.search_index(r);
    loop {
        if       (lk.height < rk.height) {
            Knoten lpk = lk.parent;
            if (lk == lpk.left_child)
                res += lpk.right_child.total_vzw;
            if (lk.max_positive != lpk.positive) res++;
            lk = lpk;
        } elseif (lk.height > rk.height) {
            Knoten rpk = rk.parent;
            if (rk == rpk.right_child)
                res += rpk.left_child.total_vzw;
            if (rk.min_positive != rpk.positive) res++;
            rk = rpk;
        } elseif (lk.heigth == rk.heigth) {
            lpk = lk.parent;
            rpk = rk.parent;
            if (lk.max_positive != lpk.positive) res++;
            if (rk.min_positive != rpk.positive) res++;
            if (lpk == rpk) break;
            if (lk == lpk.left_child)
                res += lk.right_child.total_vzw;
            if (rk == rpk.right_child)
                res += rpk.left_child.total_vzw;
            lk = lpk;
            rk = rpk;
        }
    }
    return res;
}
```

Laufzeit

Die Höhe eines AVL-Baums bleibt in $\mathcal{O}(\log n)$ und die Laufzeit von `insert` und `delete` beträgt weiterhin $\mathcal{O}(\log n)$.

Die Suche nach Index folgt dem selben Pfad wie die Suche nach Wert und hat somit eine Laufzeit in der Höhe des Baums also $\mathcal{O}(\log n)$. Das Aufsummieren der Vorzeichenwechsel der Teilbäume und auf dem Pfad ist ebenfalls linear in der Länge des Pfades also in einer Laufzeit $\mathcal{O}(\log n)$.

(Anmerkung: Auf die Elter-Zeiger könnte man auch verzichten wenn man sich den Pfad beim Abstieg in einem Stack speichert.)

Aufgabe 4B

Seien h_1 und h_2 die Höhen der AVL-Bäume T_1 und T_2. Nehmen wir an $h_2 \leq h_1$. (Für den Fall $h_1 < h_2$ gehen wir auf die gleiche Weise aber gespiegelt vor: Wir suchen statt des kleinsten Elementes aus T_2 das größte Element aus T_1 und wir betrachten den linkesten Pfad in T_2 statt des rechtesten Pfades in T_1)

Als erstes entnehmen wir das kleinste (linkeste) Element aus T_2 mit den gewöhnlichen `search` und `delete` Operationen. Das geschieht mit einer Laufzeit in $\Theta(h_2)$. Nennen wir den entstandenen AVL-Baum T_2', seine Höhe h_2' und den entnommenen Knoten s. Es gilt: $h_2 - 1 \leq h_2' \leq h_2$ und für alle Elemente $s_1 \in S_1, s_2 \in S_2 \setminus \{s\} : s_1 < s < s_2$.

Nun suchen wir auf dem rechtesten Pfad in T_1 nach dem ersten Knoten R mit einer Höhe kleiner gleich h_2'. Da der Elter von R die AVL-Eigenschaft erfüllt und eine Höhe größer h_2' besitzt, besitzt R entweder Höhe h_2' oder $h_2' - 1$. Wir ersetzen nun in T_1 den Teilbaum R durch den Baum mit Wurzel s, linkem Kind R und rechtem Kind T_2'. Dieser Baum mit Wurzel s erfüllt die Suchbaum und AVL-Eigenschaft, aber hat womöglich eine Höhe von $h_2' + 1$, wodurch die AVL-Eigenschaft am ursprünglichen Elter von r verletzt sein kann. Es genügt eine einzelne `balance` Operation um die AVL-Eigenschaft im gesamten Baum wiederherzustellen.

Für die lineare Suche nach dem Knoten r steigen wir von der Wurzel von T_1 bis zur Höhe h_2' ab. Da in jedem Schritt die Höhe um eins oder zwei sinkt, benötigen wir $\Theta(h_1 - h_2') = \Theta(h_1 - h_2)$ viele Schritte. Dabei benötigt jeder Schritt konstante Laufzeit. Das Zusammenfügen und Balancieren benötigt nur konstante Laufzeit.

Insgesamt ergibt sich also eine Laufzeit in $\mathcal{O}(h_1 - h_2) + \mathcal{O}(h_2) \subseteq \mathcal{O}(h_1)$. Man hätte die Laufzeit für beide Hälften des Algorithmus auch direkt durch $\mathcal{O}(h_1)$ abschätzen könne, so wird aber deutlich, dass der Algorithmus eine bessere Laufzeit besitzt falls man auf einen der beiden Schritte verzichten kann z.B. weil man ein Element s zwischen S_1 und S_2 erhält oder wenn die Position von r bekannt ist. Im ersten Fall ist der Algorithmus besonders schnell für Bäume ähnlicher Höhe und im zweiten Fall besonders schnell für Bäume unterschiedlicher Höhe.