Theorie

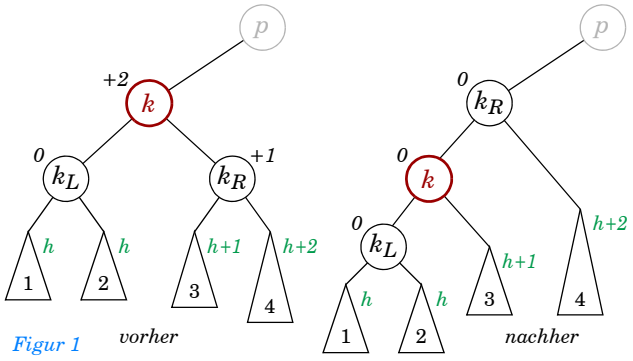
In einem gut ausbalancierten Suchbaum benötigt man $log_2(N)$ Schritte, um einen Knoten zu finden bzw. sein Nichtvorhandensein festzustellen. Unglücklicherweise führen zufällig ausgewählte Einfüge- und Lösch-Operationen in aller Regel dazu, dass der Baum entartet, d.h. seine Balance mehr oder weniger verliert, was sich ungünstig auf die Performance aller Operationen auswirkt, zumal sie alle die Suche als Hilfsoperation benötigen.

Die Lösung für dieses Problem besteht darin, nach jeder Operation, welche die Struktur des Baumes verändert, dessen Balance wieder herzustellen. Wir werden im folgenden den *AVL-Baum* näher betrachten. Dieser historisch alteste Vorschlag geht auf die Herren *Adelson*, *Velskij* und *Landis* zurück (daher auch der Name «AVL») und stammt aus dem Jahr 1962.

Ein Baum gilt als *balanciert*, wenn für jeden Knoten k des Baumes gilt: $\left|h_{k.R} - h_{k.L}\right| \leq 1$
In Worten heisst das, dass sich die Höhen der Teiläste jedes Knotens um maximal eine Ebene unterscheiden dürfen. Jeder Knoten k erhält ein zusätzliches Element, den sogenannten *Balance-Faktor* `bal`, im welchem $h_{k.R} - h_{k.L}$ gespeichert wird. Sobald in einem Knoten der Betrag des Balance-Faktors grösser als Eins geworden ist, wird eine Korrektur durchgeführt, eine *Rotation* um den betroffenen Knoten.

1. Rotationen

Mit einer Links- oder Rechts-Rotation wird das Ungleichgewicht der beiden Äste eines Knotens ausgeglichen. Die Operation an sich kann relativ einfach durch Umhängen einiger Referenzen bewerkstelligt werden. Zur Vereinfachung der Operation wird neben den in die Tiefe führenden Referenzen `L` und `R` eine weitere Referenz `U` eingeführt, welche zum Vater von k zeigt. In der Figur rechts können Sie das Ergebnis einer Links-Rotation um den Knoten k sehen.



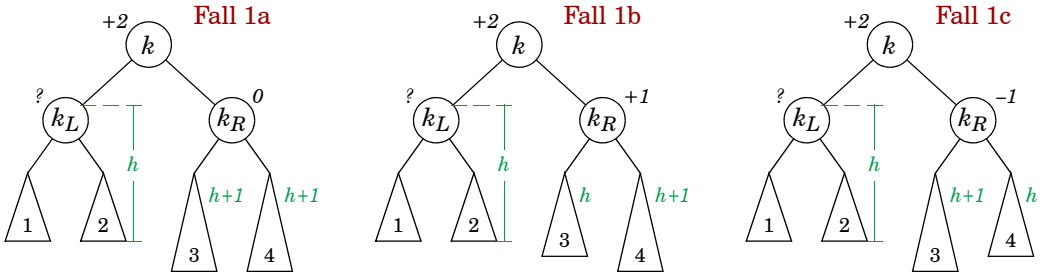
Figur 1

Es ist einfach einzusehen, dass die Grössenrelationen zwischen den Schlüsseln und somit auch die Sortier-Reihenfolge erhalten bleiben. Dieses und alle weiteren vorgestellten Rotations-Beispiele besitzen selbstverständlich spiegelbildliche Gegenstücke. Wir werden uns in diesem Text allerdings auf *Links-Rotationen* beschränken.

Eine Rotation beeinflusst die Balance-Faktoren der betroffenen Knoten k, k_L und k_R. Da sich die Höhe des Baumes unter k ändern kann, wird möglicherweise auch der Balance-Faktor von p beeinflusst. Bei einer Links-Rotation ändert sich $bal(k_L)$ garantiert nicht, während dies bei einer Rechts-Rotation für $bal(k_R)$ gilt. Generell kann man sagen, dass die *Wurzel* und der *hochgezogene Knoten* in der Regel ihre Balance ändern.

Ist es möglich, verlässliche Aussagen zu treffen, wie sich die Balance-Faktoren ändern bei einer Rotation?

Üblicherweise wird über k nur rotiert, wenn $|bal(k)| = 2$. Es gibt aber einen Sonderfall, bei dem auch für $|bal(k)| = 1$ eine Rotation durchgeführt werden muss. Wir werden diese sogenannte *Doppelrotation* weiter unten genauer untersuchen.



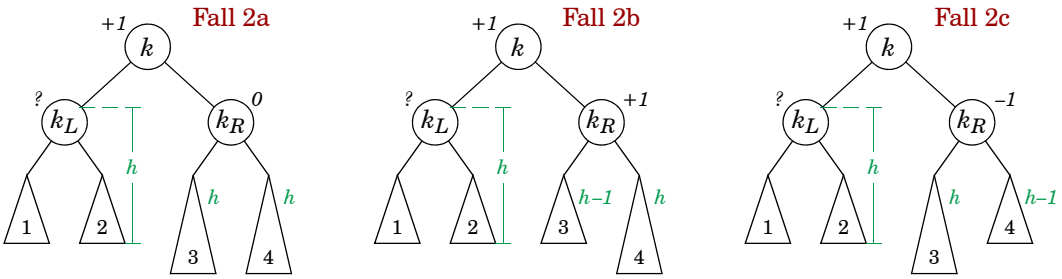
Figur 2 – Drei Fälle für die Rotation über k mit bal(k) = 2

Wir untersuchen zunächst den allgemeinen Fall der Rotation über k mit $bal(k) = 2$. Der Wert von $bal(k_L)$ spielt bei der Links-Rotation keine Rolle. Wir halten lediglich fest, dass gilt: Höhe $H(k_L) = h$. Man kann für die Fälle *1a* bis *1c* einige Kenndaten zusammentragen:

Fall	*vorher*			*nachher*			Kommentare
	$H(k)$	$bal(k)$	$bal(k_R)$	$H(k)$	$bal(k)$	$bal(k_R)$	
1a	h+3	+2	0	h+3	+1	−1	Höhe bleibt erhalten.
1b			+1	h+2	0	0	Höhe ändert sich: Beeinflusst den Vater von k.
1c			−1	h+3	+1	**−2**	Unzulässige Balance! Problem wurde nur verschoben.

Mit der ursprünglichen Balance des hochgezogenen Knotens k_R können die zukünftigen Balance-Faktoren von k und kR bestimmt werden. Fall 1c kann mit einer einfachen Rotation nicht gelöst werden. Wenn der Fall 1b aufgetreten ist, hat sich die Höhe des Teilbaumes verändert. In diesem Fall muss auch die Balance des Vaters dieses Teilbaums geprüft werden.

Wir führen die selbe Untersuchung mit Knoten durch, welche sich eigentlich noch im *«grünen Bereich»* befinden, also nur einen Balance-Faktor von +1 aufweisen:



Figur 3 – *Drei Fälle für die Rotation über k mit bal(k) = 1*

Kenndaten der Links-Rotation mit $bal(k) = 1$:

Fall	*vorher*			*nachher*			Kommentare
	$H(k)$	$bal(k)$	$bal(k_R)$	$H(k)$	$bal(k)$	$bal(k_R)$	
2a	h+2	+1	0	h+2	0	−1	Höhe ändert sich nicht. Resultat brauchbar.
2b			+1	h+2	−1	0	
2c			−1	h+2	0	**−2**	Unzulässige Balance! Problem wurde nur verschoben.

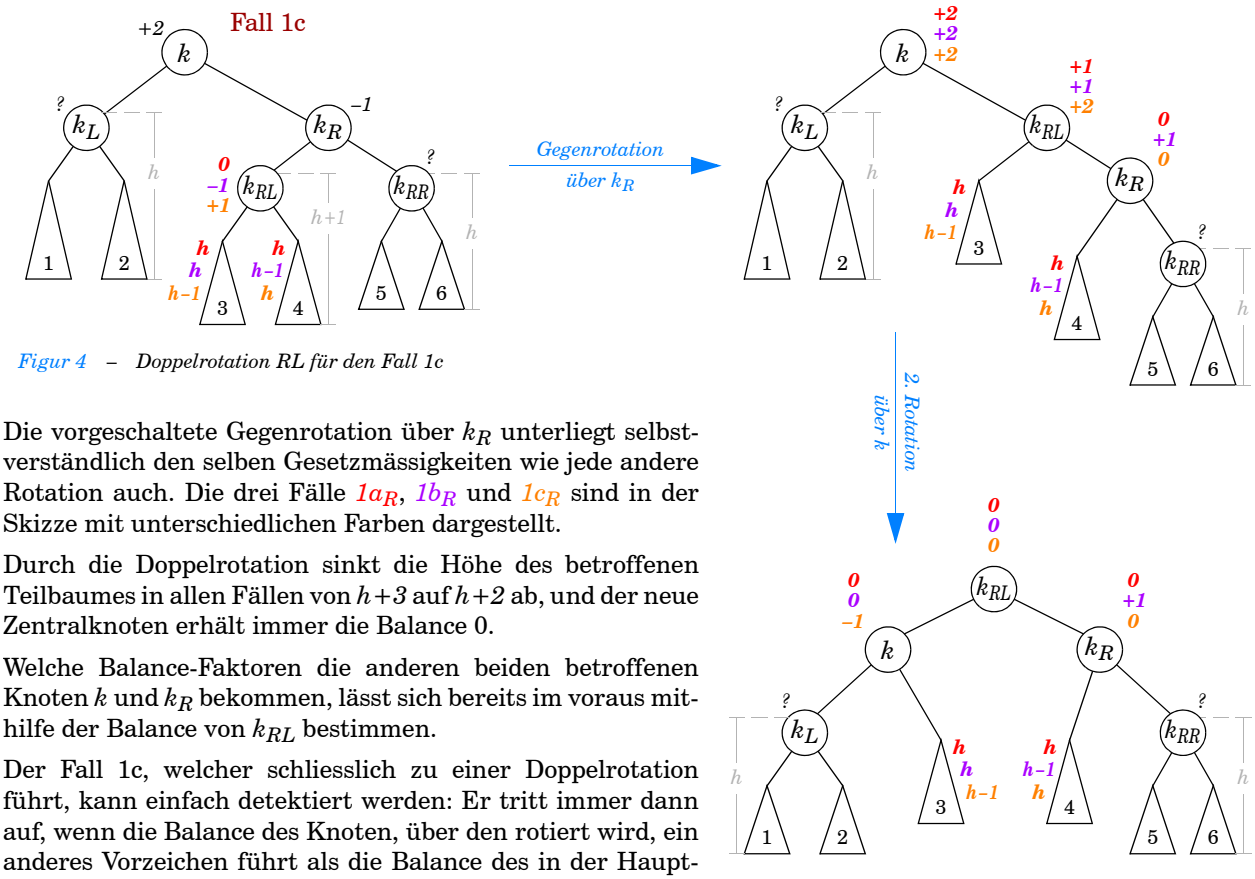
Rotationen über einem Knoten k mit $|bal(k)| = 1$ ergeben für sich keinen Sinn, da der Balance-Wert des Baumes unter k dadurch nicht verbessert werden kann. Interessant sind sie aber dennoch, weil man mit ihnen die Lastigkeit eines Teilbaumes umdrehen kann: Fälle *2a* und *2c* machen in unserem Beispiel der Linksrotation aus einem rechtslastigen einen linkslastigen Baum, Fall *2b* führt immerhin zu einer ausgeglichenen Wurzel.

Im folgenden Abschnitt wird gezeigt, wie die an sich nutzlosen Rotationen der Fallgruppe 2 verwendet werden können, um den Fall 1c zu lösen.

Vergessen Sie nicht, für alle Rotationen und Rotations-Kombinationen auch die spiegelverkehrte Situation zu analysieren.

2. Doppel-Rotationen

In Fall *1c* liegt der Baum in einer denkbar ungünstigen Konfiguration, die sich allerdings beheben lässt, indem man zuerst über den hochzuziehenden Knoten (bei Links-Rotationen k_R, bei Rechts-Rotationen k_L) eine Rotation in der Gegenrichtung ausführt und erst nachher über k rotiert. Figur 4 zeigt ein Beispiel für die Links-Rotation, wobei die Umgebung des Knotens k_R etwas detaillierter dargestellt wird:



Figur 4 – Doppelrotation RL für den Fall 1c

Die vorgeschaltete Gegenrotation über k_R unterliegt selbstverständlich den selben Gesetzmässigkeiten wie jede andere Rotation auch. Die drei Fälle $1a_R$, $1b_R$ und $1c_R$ sind in der Skizze mit unterschiedlichen Farben dargestellt.

Durch die Doppelrotation sinkt die Höhe des betroffenen Teilbaumes in allen Fällen von $h+3$ auf $h+2$ ab, und der neue Zentralknoten erhält immer die Balance 0.

Welche Balance-Faktoren die anderen beiden betroffenen Knoten k und k_R bekommen, lässt sich bereits im voraus mithilfe der Balance von k_{RL} bestimmen.

Der Fall 1c, welcher schliesslich zu einer Doppelrotation führt, kann einfach detektiert werden: Er tritt immer dann auf, wenn die Balance des Knoten, über den rotiert wird, ein anderes Vorzeichen führt als die Balance des in der Hauptrotation hochgezogenen Knotens (in der RL-Doppelrotation die Knoten k und k_R).

3. Aktualisierung der Balance-Faktoren

Wir wissen nun, wie sich Rotationen, die Grundoperationen des Balance-Ausgleichs, auf den rotierten Teilbaum auswirken: In der Mehrheit der Fälle nimmt die Höhe des Teilbaumes ab, und wir können die zukünftigen Balance-Faktoren innerhalb des Teilbaumes einfach bestimmen.

Die Änderung der Höhe eines Teilbaumes mit Wurzel k hat aber in der Regel nicht nur lokale Auswirkungen, sondern beeinflusst oft auch den Balance-Faktor des Vater-Knotens von k. Unter Umständen pflanzt sich diese Änderung weiter nach oben fort und kann sich bis in die Wurzel des gesamten Baumes auswirken.

Dabei können nicht nur *Rotationen* solche Kaskaden von Balance-Anpassungen verursachen, sondern jede beliebige einfache *Einfüge-* oder *Lösch-Operation*. Im folgenden Abschnitt untersuchen wir die Auswirkungen verschiedener Operationen auf den Balance-Haushalt eines AVL-Baumes:

a) Einfügen
b) Löschen
c) Rotationen

Wir postulieren zwei Hilfsfunktionen, die wir für die Aktualisierung der Balance-Faktoren verwenden wollen:

- `updateIn(Node p)` – Wird verwendet bei Einfüge-Operationen. Es wird der Knoten übergeben, bei dem sich eine Änderung des Balance-Faktors ergeben hat. Diese Funktion prüft, ob der Vater von p von der Änderung betroffen ist und ob sich die Änderung weiter fortpflanzt. Änderungen werden auf dem ganzen Pfad von p bis zur Wurzel verfolgt. Die Notwendigkeit einer Rotation wird ebenfalls in dieser Funktion erkannt welche die passende Operation einleitet.
- `updateOut(Node p)` – Da für das Löschen von Knoten etwas andere Bedingungen gelten als für das Einfügen, gibt es dafür eine eigene Funktion, die aber der Grundstruktur `updateIn()` ähnlich sieht.

3.1 Knoten einfügen

Neue Knoten werden grundsätzlich immer am unteren Ende des Baumes eingefügt (in dieser Hinsicht unterscheidet sich der AVL-Baum nicht vom gemeinen Binärbaum). Da vor dem Einfügen die AVL-Bedingung gilt, gibt es nur eine geringe Anzahl von Fällen, die wir zu unterscheiden haben (bei allen Einfüge-Operationen gehen wir davon aus, dass der neue Knoten direkt unter dem Knoten p eingefügt wird, und dass der Vater von p den Namen f trage):

E0 p hatte vor dem Einfügen keine Söhne. Je nachdem, ob der neue Knoten links oder rechts angehängt wird, steigt *bal(p)* auf +1 oder –1. Die Höhe des Baumes unter p steigt um 1, weshalb sich *bal(f)* ebenfalls ändern wird. `updateIn(p)` muss aufgerufen werden, um die Balance von f anzupassen und weitere Knoten auf dem Weg zur Wurzel des gesamten Baumes zu prüfen.

E1 p hatte vor dem Einfügen einen Sohn und erhält nun einen zweiten. *bal(p)* wird nun Null, da der neue Knoten den Ausgleich herbeiführt. Da sich die Höhe des Baumes unter p nicht ändert, sind keine weiteren Schritte nötig.

3.2 Knoten löschen

Vorausgesetzt, der zu löschende Knoten existiert, existieren drei Fälle, die es zu unterscheiden gilt. Es sind dies die selben, die Sie bereits in *Übung 4, Abschnitt 2.3* kennengelernt hatten. Es werden im Prinzip auch die selben Verfahren angewandt, aber es kommt der Balance-Ausgleich hinzu. Auch hier heisse der zu löschende Knoten p und dessen Vater f.

L0 p besitzt keine Söhne: Er kann ohne weiteres gelöscht werden. Je nach dem ob p linker oder rechter Sohn von f war, wird dessen Balance um 1 erhöht oder verringert. Falls sich *bal(f)* durch die Operation auf ±2 erhöht oder aber auf Null sinkt, hat sich die Höhe des Baumes über f geändert, und wir müssen `updateOut(f)` aufrufen. Andernfalls betrug vorher *bal(f)=0* und ist nun auf ±1 gewachsen: Einer seiner Äste wurde kürzer, aber die Höhe von f ändert sich nicht: Keine weiteren Aktionen nötig.

L1 p besitzt genau einen Sohn: p wird gelöscht und sein Sohn als p' an dessen Stelle gesetzt. *bal(p')* wird auf Null gesetzt. Der Baum unter p' wurde kürzer, weshalb wir `updateOut(p')` aufrufen müssen.

L2 p besitzt zwei Söhne: Wir suchen den Knoten r mit dem nächst grösseren oder nächst kleineren Schlüssel. p wird anschliessend durch r ersetzt, dann rufen die Lösch-Operation rekursiv für r auf. Für uns gibt es nichts mehr zu tun, da erst die rekursive Instanz der Lösch-Operation eine strukturelle Änderung im Baum durchführt und selbst den Ausgleich gemäss Fällen L0 bis L2 besorgt.

3.3 Balance nach dem Einfügen aktualisieren (`updateIn`)

`updateIn(p)` wird nur aufgerufen, wenn der Fall *E0* aufgetreten ist. Je nachdem ob p linker oder rechter Sohn von f ist, wird *bal(f)* dekrementiert oder inkrementiert. Anschliessend wird *bal(f)* geprüft:

- E00: *(bal(f)=0):* f wurde durch das Einfügen ausgeglichen. h_f ändert sich nicht: fertig!
- E01: $|bal(f)|=1$: h_f hat sich geändert. Neue Iteration durch `updateIn()` mit $p := f$ und $f := father(f)$
- E02: $|bal(f)|=2$: Führe eine Rotation über f aus. Nach einer einfachen Rotation ist p die Wurzel des Teilbaumes: `updateIn` wird mit $f := father(p))$ erneut iteriert, um *bal(f)* anzupassen. Nach einer Doppelrotation liegt p um eine Ebene unter der Wurzel, und es wird mit *father(p)* sowie mit *father(father(p))* erneut iteriert.

3.4 Balance nach dem Löschen aktualisieren (`updateOut`)

`updateOut(p)` wird aufgerufen, wenn der Fall *L0* oder *L1* aufgetreten ist, und zwar dann, wenn an p ein neuer Balance-Faktor von 0 oder ± 2 erkannt wurde. Da auch `updateOut(p)` als Iteration konstruiert ist, die den Pfad nötigenfalls bis zur Wurzel verfolgt, muss in der Fallunterscheidung auch ein Balance-Faktor mit Betrag 1 berücksichtigt werden:

- Lu0: *(bal(p)=0):* Die Höhe des Baumes unter p, (h_p) ist um Eins gesunken. Die Balance des Vaters f wird entsprechend angepasst. Eine weitere Iteration mit $p := f$ und $f := father(f)$ ist erforderlich.
- Lu1: $|bal(p)|=1$: Dieser Fall tritt innerhalb einer Reihe von Iterationen auf und bedeutet, dass sich die Höhe h_p nicht mehr geändert hat. Die Funktion kann also beendet werden.
- Lu2: $|bal(p)|=2$: Der Baum unter p muss rotiert werden. Je nach Konstellation von *bal(p)* und *bal(p.L)* bzw. *bal(p.R)* wird eine L- oder R-Rotation bzw. eine LR- oder eine RL-Rotation durchgeführt. Anschliessend wird mit der neuen Wurzel des rotierten Teilbaumes und deren Vater noch einmal iteriert.

4. Implementierungstips

- Die innere Knoten-Klasse muss um den Balance-Faktor erweitert werden. Es ist auch sinnvoll, eine zusätzliche Referenz einzuführen, die zum Vater eines Knotens zeigt, weil das die gesamte Implementierung wesentlich vereinfacht:

```
private static class Node{
  int key, bal;
  Node L, R, U;
}
```

- Übernehmen Sie beim Suchen die Strategie aus Übung 4 (Klasse `SearchResult`).
- Eine gegenüber Übung 4 erweiterte Version der Baum-Anzeige (mit Balance-Faktoren) könnte so aussehen:

```
public void show(){
  System.out.println();
  traverse(m_root.R, 0);     // Baum hängt hier rechts am Dummy-Knoten
}

private void traverse(Node root, int level){
  if (root != null){
    traverse(root.R, level + 1);
    for (int i = 0; i < level; ++i)
      System.out.print("        ");
    System.out.print("[");
    System.out.format("%1$03d ", root.key);
    if (root.bal != 0)
      System.out.format("%1$+2d", root.bal);
    else
      System.out.print("\u00B70");
    System.out.print("]");
    System.out.println();
    traverse(root.L, level + 1);
  }
}
```

- Es wird dringend empfohlen, die Wurzel durch ein Dummy-Element zu realisieren, da dies den Code wesentlich vereinfacht. Ob Sie Ihren Baum links oder rechts anhängen, spielt allerdings keine Rolle. Sie können den Dummy-Knoten einfach markieren, indem Sie ihn durch die U-Referenz auf sich selbst zeigen lassen.