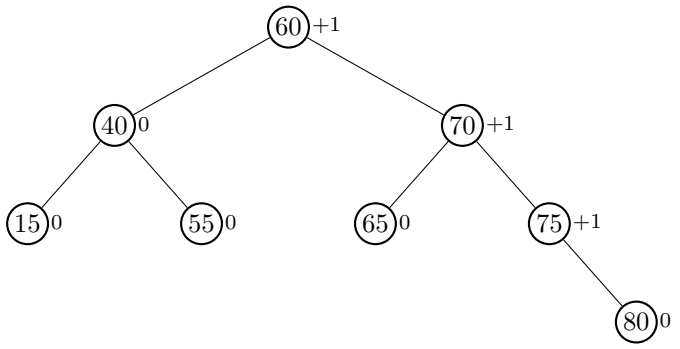
ARCHENHOLD
GYMNASIUM

Bäume

Begleitskript zum Informatikunterricht

Binärbäume · Suchbäume · Durchlaufen und Löschen · AVL-Bäume · Rotationen



Ein AVL-Baum: Die Zahl neben jedem Knoten ist seine Balance b.

Inhaltsverzeichnis

1 Bäume und Binärbäume **2**

1.1 Wo Bäume vorkommen 2

1.2 Begriffe 2

1.3 Rekursive Definition 3

2 Der Suchbaum **3**

2.1 Die Suchbaumeigenschaft 3

2.2 Einfügen 3

3 Durchlaufen eines Baums **4**

4 Löschen im Suchbaum **6**

5 Wenn der Baum entartet **7**

6 AVL-Bäume **8**

6.1 Die AVL-Eigenschaft 8

6.2 Warum das reicht: die Höhe bleibt klein 8

6.3 Rotationen 9

6.4 Einfügen in einen AVL-Baum 10

6.5 Löschen in einem AVL-Baum 11

7 Der AVL-Baum in Python: Spezifikation **13**

8 Die AVL-Simulation **14**

9 Überblick **14**

10 Ausblick: andere Bäume **16**

11 Übungen **18**

12 Lösungen **20**

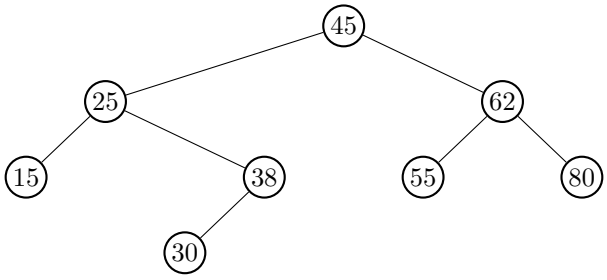
1 Bäume und Binärbäume

So arbeiten Sie mit dem Skript: Jedes Verfahren wird erklärt, an einem Beispiel durchgerechnet und als kurzer Python-Quelltext oder eine Spezifikation gezeigt (*Vor.* = Voraussetzung, *Eff.* = Effekt: Die Spezifikation sagt, *was* herauskommt, nicht *wie* es berechnet wird; die Umsetzung programmieren Sie selbst). Die Kästen *„auf einen Blick“* fassen ein Verfahren zusammen. Abschnitt 11 enthält Übungen, Abschnitt 12 die vollständigen Lösungen: erst selbst zeichnen, dann vergleichen. Alle Bäume, Zahlen und Quelltexte in diesem Skript sind mit einem Programm nachgerechnet. Mit der *AVL-Simulation* (Abschnitt 8) können Sie jedes Beispiel selbst nachbauen.

1.1 Wo Bäume vorkommen

Ordner in einem Dateisystem, ein Stammbaum, die Gliederung dieses Skripts, ein Rechenterm wie $(3 + 4) \cdot 5$ – überall ist etwas *hierarchisch* aufgebaut: Ein Element enthält oder erzeugt weitere Elemente. In der Informatik heißt eine solche Struktur *Baum*. In diesem Skript betrachten wir *Binärbäume*: Jeder Knoten hat höchstens zwei Kinder.

1.2 Begriffe



Ein Binärbaum mit 8 Knoten (Wurzel 45)

Wurzel: der oberste Knoten (45). Jeder andere Knoten hat genau einen *Elternknoten*.

Kind, Elternknoten: 25 und 62 sind die Kinder von 45; 45 ist ihr Elternknoten. Beim Binärbaum gibt es ein *linkes* und ein *rechtes* Kind.

Blatt, innerer Knoten: Ein Blatt hat keine Kinder (15, 30, 55, 80), alle anderen heißen *innere Knoten*.

Teilbaum: ein Knoten mit allem, was unter ihm hängt. Der linke Teilbaum der Wurzel ist der Baum mit der Wurzel 25.

Pfad, Tiefe: Die *Tiefe* eines Knotens ist die Länge des Pfades von der Wurzel zu ihm (die Wurzel hat Tiefe 0; 30 hat Tiefe 3).

Höhe: Die *Höhe* eines Baums ist die größte Tiefe eines Knotens, also die Länge des längsten Pfades von der Wurzel zu einem Blatt. Der Baum oben hat die Höhe 3. Ein einzelner Knoten hat die Höhe 0, der *leere* Baum die Höhe -1.

1.3 Rekursive Definition

Ein Binärbaum ist entweder *leer* oder er besteht aus einem *Knoten* mit einem Wert, einem linken und einem rechten *Teilbaum*, die selbst wieder Binärbäume sind. Diese Definition passt genau zu den rekursiven Funktionen, die wir schreiben: Was für den ganzen Baum gilt, ergibt sich aus dem, was für die beiden Teilbäume gilt. Die Höhe zum Beispiel:

$$h(\text{leer}) = -1, \qquad h(\text{Knoten}) = 1 + \max\big(h(\text{links}),\, h(\text{rechts})\big).$$

Quelltext 1: Ein Knoten und die Höhe eines Baums (der leere Baum ist `None`)

```
class Knoten:
    def __init__(self, wert):
        self.wert = wert
        self.links = None          # linker Teilbaum (None = leer)
        self.rechts = None         # rechter Teilbaum

def hoehe(k):
    if k is None:
        return -1                  # leerer Baum
    return 1 + max(hoehe(k.links), hoehe(k.rechts))
```

2 Der Suchbaum

2.1 Die Suchbaumeigenschaft

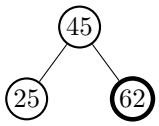
Ein Binärbaum wird zum *Suchbaum*, wenn für jeden Knoten gilt:

> Alle Werte im **linken** Teilbaum sind **kleiner**, alle Werte im **rechten** Teilbaum sind **größer** als der Wert des Knotens.

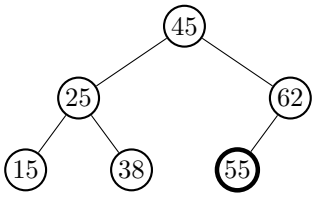
Dann muss man beim Suchen nicht mehr alles prüfen: Ist die gesuchte Zahl kleiner als der Wert im Knoten, geht es links weiter, sonst rechts. Bei jedem Schritt wird die Hälfte des Baums ausgeschlossen – wenn der Baum *gleichmäßig* gebaut ist (Abschnitt 5).

2.2 Einfügen

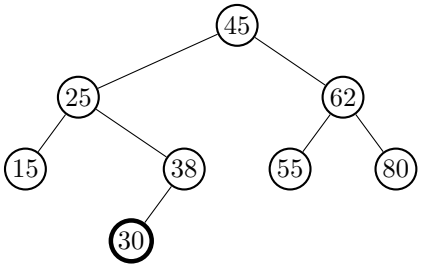
Ein neuer Wert wird immer als *Blatt* eingefügt: Man sucht die Stelle, an der der Wert stehen müsste, und hängt ihn dort an. Beispiel: Die Zahlen 45, 25, 62, 15, 38, 55, 80, 30 werden in dieser Reihenfolge in einen leeren Suchbaum eingefügt.



nach 45, 25, 62 (zuletzt: 62)



nach den ersten 6 Zahlen (zuletzt: 55)



nach den ersten 8 Zahlen (zuletzt: 30)

Der Baum hängt von der *Reihenfolge* ab: Dieselben Zahlen in anderer Reihenfolge ergeben einen anderen Baum.

Quelltext 2: Einfügen im Suchbaum (Wiederholungen werden nicht eingefügt); Suchen als Spezifikation

```
def einfuegen(k, wert):            # liefert die (neue) Wurzel des Teilbaums
    if k is None:
        return Knoten(wert)        # freie Stelle gefunden: neues Blatt
    if wert < k.wert:
        k.links = einfuegen(k.links, wert)
    elif wert > k.wert:
        k.rechts = einfuegen(k.rechts, wert)
    return k                       # wert schon vorhanden: nichts tun

wurzel = None
for z in [45, 25, 62, 15, 38, 55, 80, 30]:
    wurzel = einfuegen(wurzel, z)

def suchen(k, wert):
    """ Vor.:  k ist die Wurzel eines Suchbaums (None = leerer Baum)
        Eff.:  liefert True, wenn wert im Baum vorkommt, sonst False """
```

Suchbaum auf einen Blick	
Idee	Links kleiner, rechts größer; neue Werte werden Blätter
Ablauf	Von der Wurzel aus vergleichen und nach links oder rechts gehen, bis die Stelle frei ist (oder der Wert gefunden wird)
Aufwand	höchstens so viele Schritte wie die Höhe des Baums
Vorsicht	Die Höhe hängt von der Reihenfolge ab (Abschnitt 5)

3 Durchlaufen eines Baums

Um alle Knoten zu besuchen, gibt es vier übliche Reihenfolgen. Die ersten drei unterscheiden sich nur darin, *wann* die Wurzel eines Teilbaums besucht wird:

Name	Reihenfolge	rekursiv
Preorder	Wurzel, links, rechts	erst Wurzel, dann beide Teilbäume
Inorder	links, Wurzel, rechts	linker Teilbaum, Wurzel, rechter Teilbaum
Postorder	links, rechts, Wurzel	beide Teilbäume, dann Wurzel
Levelorder	Ebene für Ebene	mit einer Schlange (von oben nach unten, links nach rechts)

Für den Baum aus Abschnitt 2 (Wurzel 45) ergibt sich:

Durchlauf	besuchte Knoten
Preorder	45, 25, 15, 38, 30, 62, 55, 80
Inorder	15, 25, 30, 38, 45, 55, 62, 80
Postorder	15, 30, 38, 25, 55, 80, 62, 45
Levelorder	45, 25, 62, 15, 38, 55, 80, 30

Die **Inorder**-Reihenfolge eines Suchbaums liefert die Werte **sortiert**. Der Grund ist die Suchbaumeigenschaft: Erst kommt alles, was kleiner als die Wurzel ist, dann die Wurzel, dann alles, was größer ist – und in den Teilbäumen gilt das Gleiche.

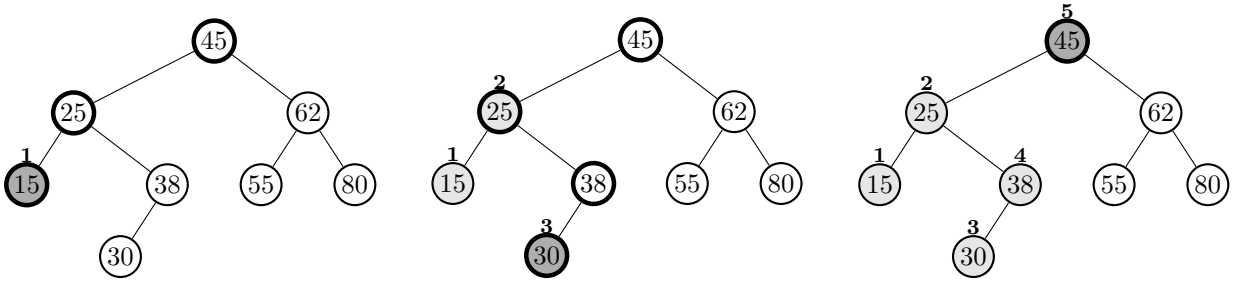
Tiefendurchlauf und Breitendurchlauf

Die vier Reihenfolgen gehören zu zwei grundsätzlich verschiedenen Strategien:

- **Tiefendurchlauf** (*Preorder, Inorder, Postorder*): Man verfolgt einen Ast bis ganz nach unten, kehrt dann zurück und nimmt den nächsten Ast. Das ist genau die Rekursion: Jeder Aufruf merkt sich, wo es nach der Rückkehr weitergeht. Dieser Merkzettel ist der Aufrufstapel, also ein *Stapel* (Stack, zuletzt hinein, zuerst heraus). Man kann die Rekursion auch durch einen eigenen Stapel ersetzen.
- **Breitendurchlauf** (*Levelorder*): Man besucht erst alle Knoten einer Ebene, bevor man die nächste beginnt. Dazu braucht man eine *Schlange* (Queue, zuerst hinein, zuerst heraus): Die Kinder eines besuchten Knotens stellen sich hinten an. Deshalb kommt im Quelltext keine Rekursion vor.

Tiefe ↔ Rekursion bzw. **Stapel**, Breite ↔ **Schlange**. Die Strategie ändert sich nur durch die Datenstruktur, in der die noch offenen Knoten warten.

Tiefendurchlauf (Inorder) mit Stapel. Die Zahl über einem Knoten ist seine Besuchsnummer. Grau: der aktuelle Knoten; hellgrau: schon besucht; dicker Rand: Knoten, deren Aufruf noch offen ist (sie stehen im Stapel, zuletzt aufgerufen ganz rechts).

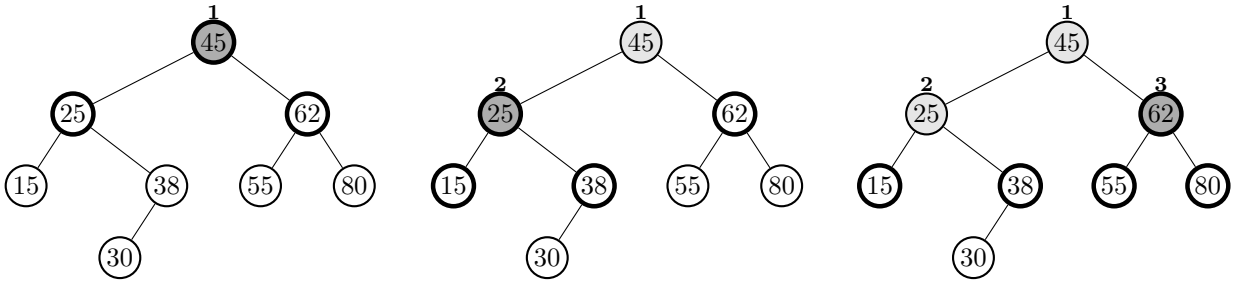


besucht: 15
Stapel: 45 → 25 → 15

besucht: 15, 25, 30
Stapel: 45 → 25 → 38 → 30

besucht: 15, 25, 30, 38, 45
Stapel: 45

Breitendurchlauf (Levelorder) mit Schlange. Der dicke Rand zeigt die Knoten, die in der Schlange warten (vorn links). Wer vorn herausgenommen wird, wird besucht; seine Kinder stellen sich hinten an.



besucht: 45
Schlange: 25 | 62

besucht: 45, 25
Schlange: 62 | 15 | 38

besucht: 45, 25, 62
Schlange: 15 | 38 | 55 | 80

Dieselben Abläufe zeigt die Simulation (Abschnitt 8) Schritt für Schritt.

Bei Graphen heißen die gleichen Verfahren *Tiefensuche* (DFS) und *Breitensuche* (BFS). Dort wird meist ein bestimmter Knoten *gesucht* und der Durchlauf dann abgebrochen; im Baum besuchen wir alle Knoten und sprechen deshalb von *Durchlauf*. Bei Graphen kommt außerdem hinzu, dass man sich besuchte Knoten merken muss, weil es Kreise geben kann.

Quelltext 3: Durchläufe: Preorder und Levelorder als Quelltext, Inorder und Postorder als Spezifikation (die Ergebnisse stehen in einer Liste)

```
def preorder(k, liste):
    if k is not None:
        liste.append(k.wert)       # erst die Wurzel
        preorder(k.links, liste)
        preorder(k.rechts, liste)

def inorder(k, liste):
    """ Vor.:  k ist die Wurzel eines Suchbaums (None = leerer Baum)
               liste ist eine Liste
        Eff.:  liste enthält zusätzlich alle Werte des Baums
               in der Inorder-Reihenfolge """

def postorder(k, liste):
    """ Vor.:  k ist die Wurzel eines Suchbaums (None = leerer Baum)
               liste ist eine Liste
        Eff.:  liste enthält zusätzlich alle Werte des Baums
               in der Postorder-Reihenfolge """

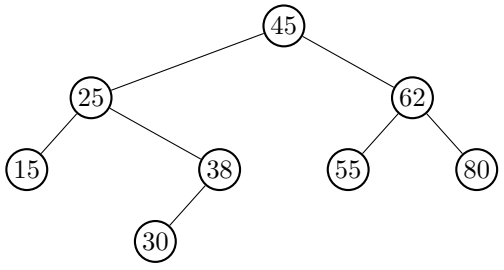
def levelorder(wurzel):
    liste = []
    schlange = [wurzel]
    while len(schlange) > 0:
        k = schlange.pop(0)        # vorderstes Element
        liste.append(k.wert)
        if k.links is not None:
            schlange.append(k.links)
        if k.rechts is not None:
            schlange.append(k.rechts)
    return liste
```

4 Löschen im Suchbaum

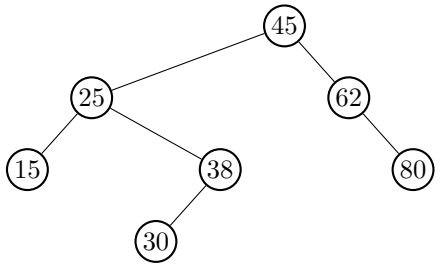
Beim Löschen darf die Suchbaumeigenschaft nicht verloren gehen. Es gibt drei Fälle:

1. Der Knoten ist ein **Blatt**: einfach entfernen.
2. Der Knoten hat **ein Kind**: Das Kind rückt an seine Stelle.
3. Der Knoten hat **zwei Kinder**: Er wird durch das **Maximum seines linken Teilbaums** ersetzt (den größten Wert, der noch kleiner ist; man geht einmal nach links und dann immer nach rechts). Dieses Maximum hat kein rechtes Kind und wird an seiner alten Stelle entfernt (Fall 1 oder 2).

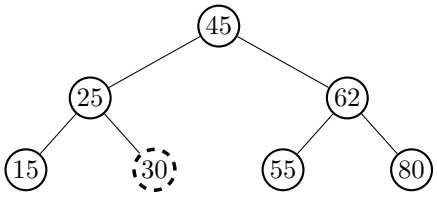
Beispiel: Aus dem Baum von Abschnitt 2 werden nacheinander 55 (ein Blatt), 38 (ein Kind) und 45 (zwei Kinder) *jeweils aus dem ursprünglichen Baum* gelöscht.



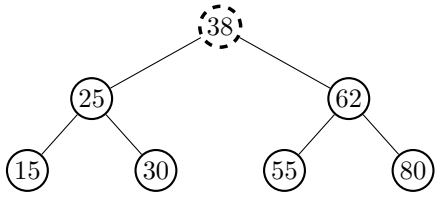
Ausgangsbaum



55 gelöscht (Blatt)



38 gelöscht (ein Kind: 30 rückt nach)



45 gelöscht (zwei Kinder: das Maximum 38 des linken Teilbaums ersetzt 45; 30 rückt nach)

Als Python-Funktionen sind die drei Fälle so zu spezifizieren (den Quelltext untersuchen Sie in Aufgabe 4):

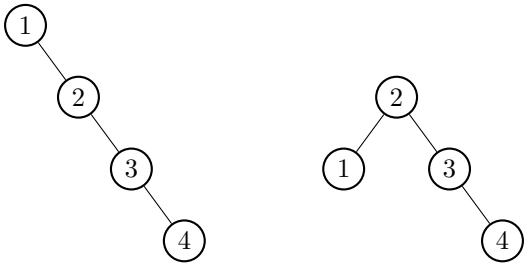
Quelltext 4: Löschen im Suchbaum: Spezifikation

```
def maximum(k):
    """ Vor.:  k ist die Wurzel eines nichtleeren Suchbaums
        Eff.:  liefert den Knoten mit dem größten Wert des Baums """

def loeschen(k, wert):
    """ Vor.:  k ist die Wurzel eines Suchbaums (None = leerer Baum)
        Eff.:  liefert die Wurzel des Suchbaums, aus dem wert entfernt ist
               (kommt wert nicht vor, ist der Baum unverändert);
               bei einem Knoten mit zwei Kindern nimmt das Maximum
               des linken Teilbaums dessen Platz ein """
```

5 Wenn der Baum entartet

Fügt man die Zahlen $1, 2, 3, \ldots$ in dieser Reihenfolge in einen Suchbaum ein, wird jede neue Zahl das rechte Kind der vorigen. Der Baum ist zu einer *Liste* entartet: Suchen braucht im schlimmsten Fall so viele Schritte, wie der Baum Knoten hat.



Dieselben vier Zahlen: einmal sortiert eingefügt (Höhe 3), einmal in der Reihenfolge 2, 1, 3, 4 (Höhe 2)

Wie stark der Unterschied bei vielen Zahlen ist, zeigt eine Messung (jeweils ohne Ausgleich eingefügt; die zufälligen Reihenfolgen sind Mittelwerte aus 200 Versuchen):

n	sortiert eingefügt	zufällig eingefügt	AVL (sortiert eingefügt)	kleinstmöglich
15	14	5,9	3	3
127	126	13,2	6	6
1023	1022	21,1	9	9

Die letzte Spalte zeigt die kleinste Höhe, die ein Binärbaum mit n Knoten überhaupt haben kann ($\lfloor \log_2 n \rfloor$). Sortierte Eingaben sind in der Praxis häufig (alphabetische Listen, Messreihen, Zeitstempel) – genau dann ist ein gewöhnlicher Suchbaum am schlechtesten. Wir brauchen einen Baum, der sich *selbst ausgleicht*.

6 AVL-Bäume

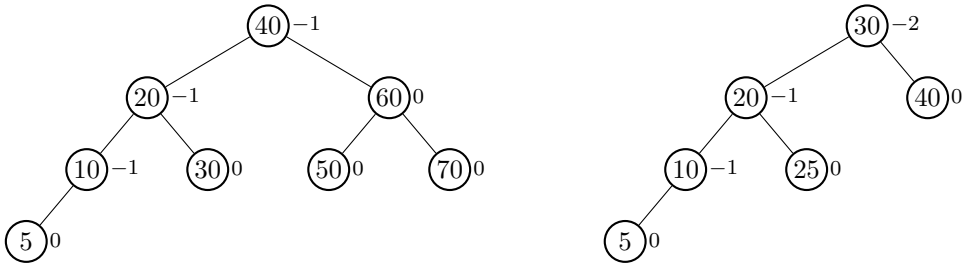
6.1 Die AVL-Eigenschaft

Der AVL-Baum (benannt nach Adelson-Velskij und Landis, 1962) ist ein Suchbaum, der nach jedem Einfügen und Löschen wieder *ausgeglichen* wird. Dazu betrachtet man an jedem Knoten k die *Balance*

$$b(k) = h(\text{rechter Teilbaum von } k) - h(\text{linker Teilbaum von } k).$$

Ein Suchbaum ist ein **AVL-Baum**, wenn für **jeden** Knoten k gilt: $b(k) \in \{-1, 0, +1\}$ – die Höhen der beiden Teilbäume unterscheiden sich um höchstens 1.

Positive Balance heißt: rechts ist es höher („rechtslastig“), negative: links ist es höher.



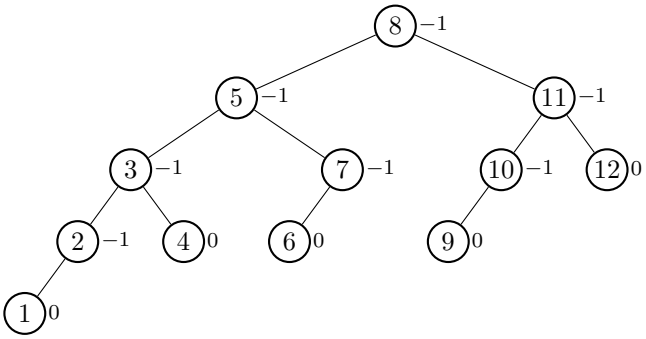
Links ein AVL-Baum, rechts ein Suchbaum, der keiner ist (die Zahl neben dem Knoten ist die Balance b; bei 30 ist $b = -2$)

6.2 Warum das reicht: die Höhe bleibt klein

Wie viele Knoten braucht ein AVL-Baum der Höhe h mindestens? Der *dünnste* AVL-Baum der Höhe h hat einen Teilbaum der Höhe $h-1$ und einen der Höhe $h-2$ (mehr Unterschied ist nicht erlaubt), beide wieder so dünn wie möglich. Mit $N(h)$ als Mindestzahl der Knoten gilt

$$N(0) = 1, \quad N(1) = 2, \quad N(h) = N(h-1) + N(h-2) + 1.$$

Höhe h	0	1	2	3	4	5	6	7	8
$N(h)$	1	2	4	7	12	20	33	54	88



Der dünnste AVL-Baum der Höhe 4 hat 12 Knoten

Die Zahlen wachsen etwa wie die Fibonacci-Zahlen, also *exponentiell*: Pro zusätzlicher Ebene wird die Mindestknotenzahl ungefähr mit dem gleichen Faktor multipliziert. Umgekehrt wächst die Höhe nur *logarithmisch* mit der Knotenzahl, ähnlich wie bei einem vollständigen Baum ($2^h \leftrightarrow \log_2 n$): Verdoppelt sich die Zahl der Knoten, kommt nur etwa eine Ebene dazu. Bei einer Million Knoten ist die Höhe höchstens 28 (ein vollständiger Baum käme mit 19 aus). Suchen, Einfügen und Löschen brauchen also nie mehr als etwa 28 Schritte statt einer Million.

Genauer: Die Höhe ist höchstens $1{,}44 \cdot \log_2(n+2)$. Der Faktor 1,44 stammt aus dem Fibonacci-Wachstum; die Herleitung brauchen wir nicht. Merken Sie sich: Die Höhe ist $\approx \log_2 n$, im ungünstigsten Fall höchstens etwa das 1,44-Fache des besten Falls.

6.3 Rotationen

Nach dem Einfügen (oder Löschen) kann die Balance eines Knotens auf $+2$ oder -2 springen. Dann wird der Teilbaum durch eine *Rotation* umgebaut, ohne die Suchbaumeigenschaft zu verletzen. Entscheidend sind die Balance des Knotens k und die Balance seines schwereren Kindes:

$b(k)$	Balance des Kindes	Fall	Behebung
$+2$	≥ 0 (rechts-rechts)	RR	**Linksrotation** um k
$+2$	< 0 (rechts-links)	RL	erst **Rechtsrotation** um das rechte Kind, dann **Linksrotation** um k
-2	≤ 0 (links-links)	LL	**Rechtsrotation** um k
-2	> 0 (links-rechts)	LR	erst **Linksrotation** um das linke Kind, dann **Rechtsrotation** um k

Die Dreiecke stehen für beliebige Teilbäume A, B, C, D; sie behalten ihre Gestalt und werden nur woanders angehängt. Weil $A < k < B < r < C$ gilt, bleibt die Inorder-Reihenfolge – und damit die Suchbaumeigenschaft – erhalten.

Einfache Rotationen



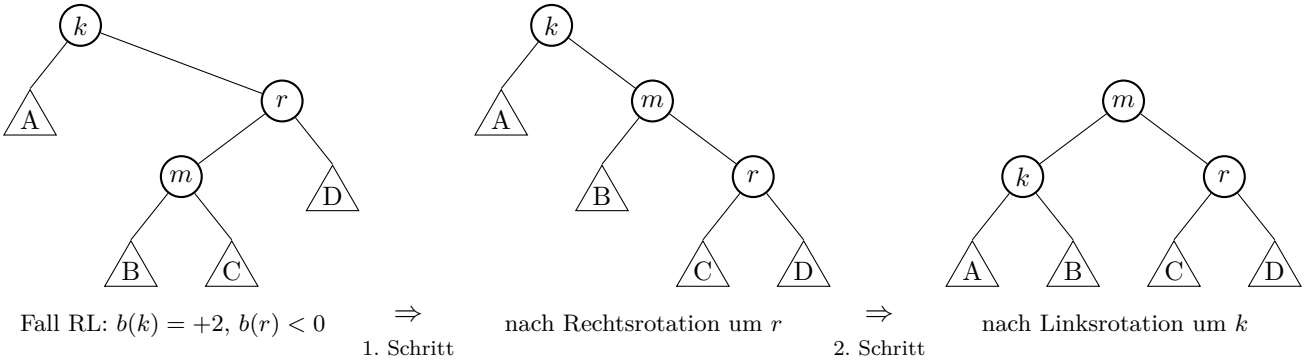
Fall RR: $b(k) = +2$ ⇒ Linksrotation um k nach der Linksrotation um k

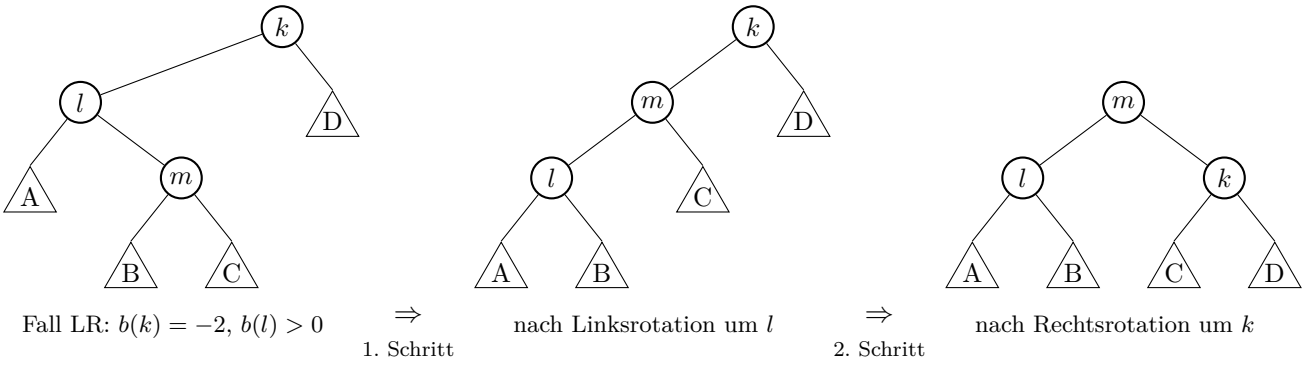


Fall LL: $b(k) = -2$ ⇒ Rechtsrotation um k nach der Rechtsrotation um k

Doppelrotationen

Ist das schwerere Kind in die *andere* Richtung schwer (Fall RL oder LR), reicht eine einzelne Rotation nicht – der mittlere Teilbaum bliebe auf der falschen Seite. Dann dreht man zuerst um das Kind und danach um k:





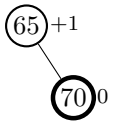
AVL-Ausgleich auf einen Blick	
Idee	Nach jeder Änderung die Balance prüfen und bei $\|b\| = 2$ rotieren
Ablauf	Von der geänderten Stelle aufwärts zur Wurzel gehen, an jedem Knoten b bestimmen; bei ± 2: Fall bestimmen (RR, LL, RL, LR) und rotieren
Aufwand	Einfügen: höchstens eine (Doppel-)Rotation; Löschen: möglicherweise eine je Ebene; insgesamt $O(\log n)$
Vorsicht	Beim Löschen kann das Kind die Balance 0 haben – dann genügt die einfache Rotation (Fall RR bzw. LL)

6.4 Einfügen in einen AVL-Baum

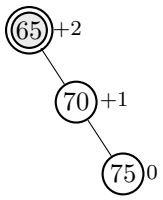
Beispiel: Die Zahlen 65, 70, 75, 15, 60, 55, 40, 80 werden nacheinander eingefügt. Die Bilder zeigen den Baum nach dem Einfügen jeder Zahl und – wenn eine Rotation nötig war – den Zustand *vor* der Rotation (doppelt umrandet: der Knoten mit $|b| = 2$) und *danach*.



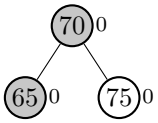
Einfügen von 65



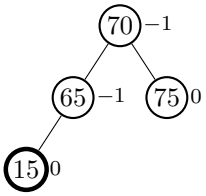
Einfügen von 70



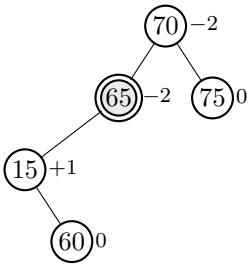
nach dem Einfügen von 75:
$b(65) = +2$, Fall RR



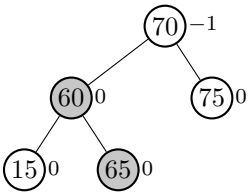
nach der Linksrotation um 65



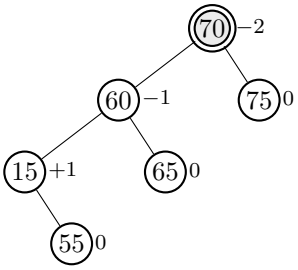
Einfügen von 15



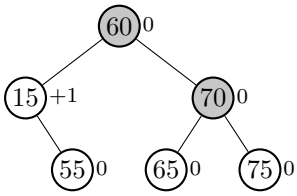
nach dem Einfügen von 60:
$b(65) = -2$, Fall LR



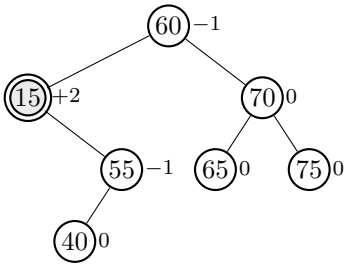
nach der
Links-Rechts-Doppelrotation um 65



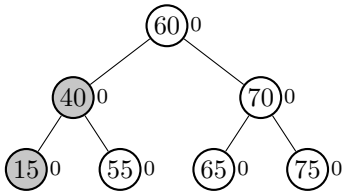
nach dem Einfügen von 55:
$b(70) = -2$, Fall LL



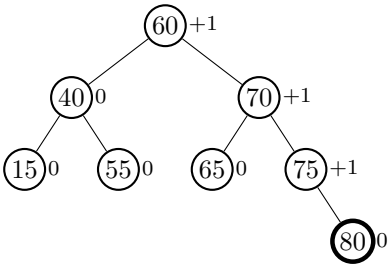
nach der Rechtsrotation um 70



nach dem Einfügen von 40:
$b(15) = +2$, Fall RL



nach der
Rechts-Links-Doppelrotation um 15



Einfügen von 80

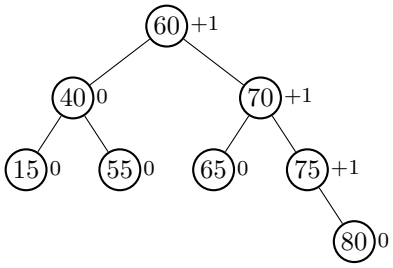
Alle vier Fälle kommen vor: RR (beim Einfügen von 75), LR (beim Einfügen von 60), LL (beim Einfügen von 55), RL (beim Einfügen von 40).

6.5 Löschen in einem AVL-Baum

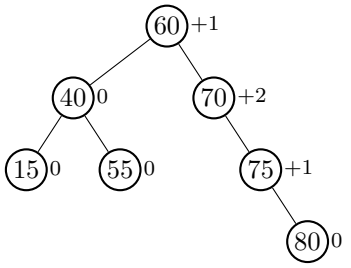
Balance $b(k) = h(\text{rechts}) - h(\text{links})$. Bei $b = +2$ ist der Knoten rechtslastig, bei $b = -2$ linkslastig. Erlaubt sind nur -1, 0, $+1$.

Gelöscht wird zuerst wie im Suchbaum (Abschnitt 4). Danach geht man vom Ort der Änderung bis zur Wurzel und gleicht jeden Knoten mit $|b| = 2$ aus. Anders als beim Einfügen kann das *mehrmals* nötig sein, weil eine Rotation die Höhe des Teilbaums verkleinern kann.

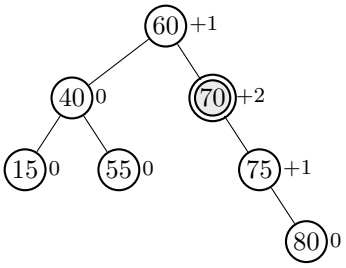
Beispiel 1: Aus dem Baum aus Abschnitt 6.4 wird die 65 gelöscht (ein Blatt). Der Knoten 70 hat danach $b = +2$ und sein rechtes Kind 75 hat $b = +1$: Fall RR, also Linksrotation um 70.



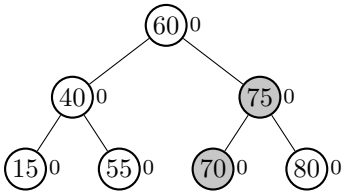
Ausgangsbaum



65 gelöscht, noch ohne Ausgleich

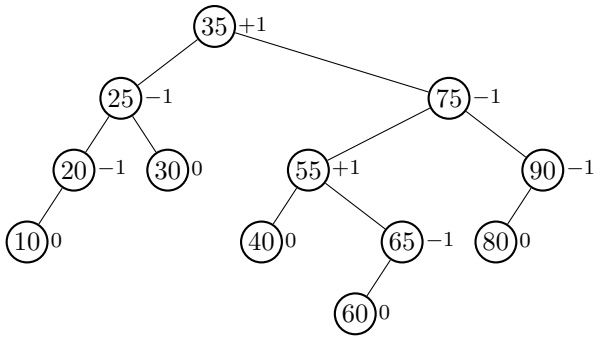


$b(70) = +2$, Fall RR

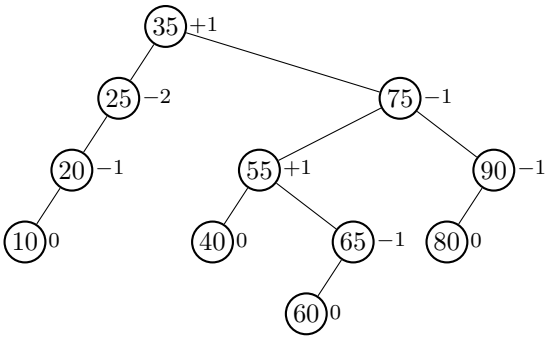


nach der Linksrotation um 70

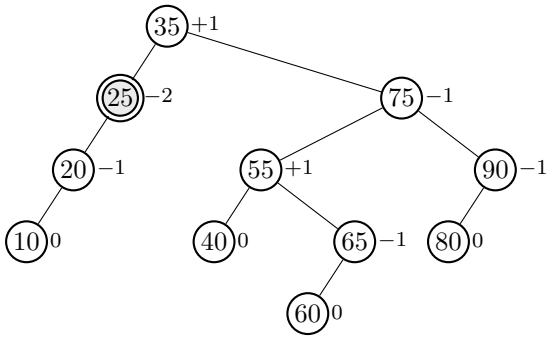
Beispiel 2 (zwei Ausgleiche hintereinander): Die Zahlen 20, 10, 25, 75, 90, 65, 80, 30, 35, 40, 55, 60 werden eingefügt (mit Ausgleich) und danach die 30 gelöscht (ein Blatt). Trotzdem muss danach *zweimal* ausgeglichen werden: zuerst Fall LL, dann weiter oben Fall RL. Die Doppelrotation ist dabei in ihre beiden Einzelschritte zerlegt.



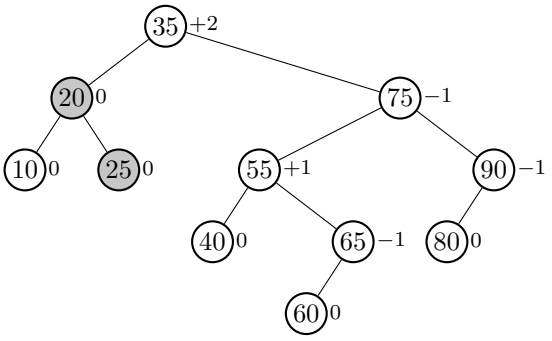
Ausgangsbaum



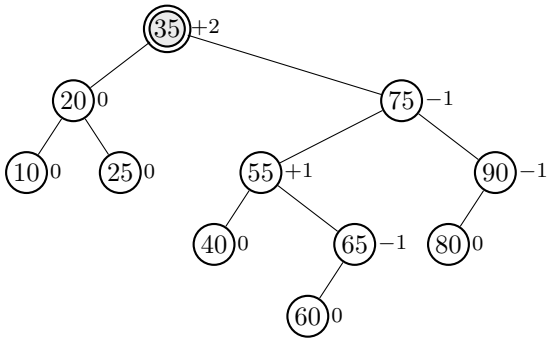
30 gelöscht, noch ohne Ausgleich



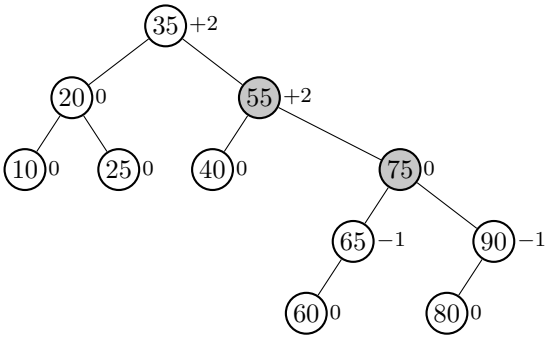
$b(25) = -2$, Fall LL



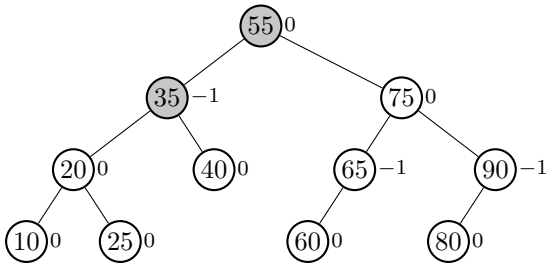
nach der Rechtsrotation um 25



$b(35) = +2$, Fall RL



1. Schritt: Rechtsrotation um 75



2. Schritt: Linksrotation um 35 (fertig)

7 Der AVL-Baum in Python: Spezifikation

Der AVL-Baum ergänzt den Suchbaum um *Rotationen* und einen *Ausgleich*; Knoten, `hoehe` und `maximum` bleiben wie bisher. Die Umsetzung ist Aufgabe der Arbeitsblätter, hier steht die Spezifikation:

Quelltext 5: AVL-Baum: Spezifikation der Funktionen

```
def balance(k):
    """ Vor.:  k ist ein Knoten (nicht None)
        Eff.:  liefert Höhe des rechten minus Höhe des linken Teilbaums von k """

def lrot(k):
    """ Vor.:  k hat ein rechtes Kind
        Eff.:  liefert die neue Wurzel nach Linksrotation um k; Inorder-Folge unverändert """

def rrot(k):
    """ Vor.:  k hat ein linkes Kind
        Eff.:  liefert die neue Wurzel nach Rechtsrotation um k; Inorder-Folge unverändert """

def ausgleichen(k):
    """ Vor.:  k ist Wurzel eines Suchbaums mit AVL-Teilbäumen (Höhenunterschied bis 2 erlaubt)
        Eff.:  liefert die Wurzel eines AVL-Baums mit denselben Werten wie der Baum k """

def einfuegen(k, wert):
    """ Vor.:  k ist die Wurzel eines AVL-Baums (None = leerer Baum)
        Eff.:  liefert die Wurzel eines AVL-Baums mit allen Werten von k und zusätzlich wert
               (ein schon vorhandener Wert wird nicht noch einmal eingefügt) """

def loeschen(k, wert):
    """ Vor.:  k ist die Wurzel eines AVL-Baums (None = leerer Baum)
        Eff.:  liefert die Wurzel eines AVL-Baums mit allen Werten von k außer wert;
               bei zwei Kindern nimmt das Maximum des linken Teilbaums den Platz ein """
```

Zur Kontrolle. Ihre Umsetzung ist richtig, wenn nach jedem Einfügen und Löschen

- die Inorder-Reihenfolge sortiert ist (die Suchbaumeigenschaft gilt),
- die Balance *jedes* Knotens in $\{-1, 0, +1\}$ liegt,
- und derselbe Baum entsteht wie in der AVL-Simulation (Abschnitt 8) bei gleicher Eingabe.

8 Die AVL-Simulation

Zu diesem Skript gehört ein Programm, das den AVL-Baum Schritt für Schritt aufbaut (Python mit `tkinter`; eine Web-Fassung für die Schulseite ist geplant). Oben gibt man Zahlen ein, unten entsteht der Baum. Neben jedem Knoten steht die Balance b; verletzt ein Knoten das AVL-Kriterium, wird er rot markiert, und die folgende Rotation wird angezeigt.

- **Einfügen / Löschen:** mehrere Zahlen durch Leerzeichen oder Komma trennen; *Enter* fügt ein, *Umschalt+Enter* löscht.
- **Schritt für Schritt:** mit *Weiter* und *Zurück* (oder den Pfeiltasten) einzelne Schritte ansehen; die Knoten gleiten beim Rotieren an ihren neuen Platz.
- **AVL-Ausgleich ausschalten:** Dann entsteht ein gewöhnlicher Suchbaum. So sieht man, wie ein sortiert eingegebener Baum entartet (Abschnitt 5).
- **Durchlauf:** Die Knöpfe *Preorder*, *Inorder*, *Postorder* (Tiefendurchlauf) und *Levelorder* (Breitendurchlauf) lassen eine Markierung Schritt für Schritt durch den Baum laufen. Darunter wächst die Folge der besuchten Schlüssel; beim Tiefendurchlauf zeigt die Anzeige zusätzlich den Stapel der offenen Aufrufe, beim Breitendurchlauf die Schlange.
- **Beispiele:** Die Auswahl enthält jeden der vier Fälle, die Beispiele dieses Skripts und drei Löschbeispiele.

Experimente.

1. Fügen Sie $1, 2, 3, 4, 5, 6, 7$ ein – einmal mit, einmal ohne Ausgleich. Welche Höhen entstehen?
2. Fügen Sie die Zahlen 10, 20, 30, 40, 50, 25 ein. Welche Fälle treten auf?
3. Bauen Sie einen Baum mit 15 Zufallszahlen und löschen Sie die Wurzel. Wer rückt nach, und wird rotiert?

9 Überblick

	Suchbaum (ohne Ausgleich)	AVL-Baum
Höhe, ungünstigster Fall	$n - 1$ (Liste)	$\approx \log_2 n$
Suchen	höchstens Höhe + 1 Vergleiche, im ungünstigsten Fall n	höchstens $\approx \log_2 n$ Vergleiche
Einfügen	wie Suchen	Suchen + Ausgleich (höchstens 1 Rotation)
Löschen	wie Suchen	Suchen + Ausgleich (höchstens 1 Rotation je Ebene)
Inorder-Durchlauf	sortierte Folge	sortierte Folge

- Der Suchbaum ist einfach, aber seine Leistung hängt von der Reihenfolge der Eingabe ab.
- Der AVL-Baum *garantiert* eine logarithmische Höhe – auch bei sortierter Eingabe. Der Preis sind die Rotationen und das Mitführen der Balance.
- Die Rotationen ändern die Form, nie die Inorder-Folge: Ein AVL-Baum ist immer ein korrekter Suchbaum.
- Andere ausgeglichene Bäume (Rot-Schwarz-Bäume, B-Bäume) beruhen auf demselben Gedanken; was sie anders machen, steht im Ausblick (Abschnitt 10).

10 Ausblick: andere Bäume

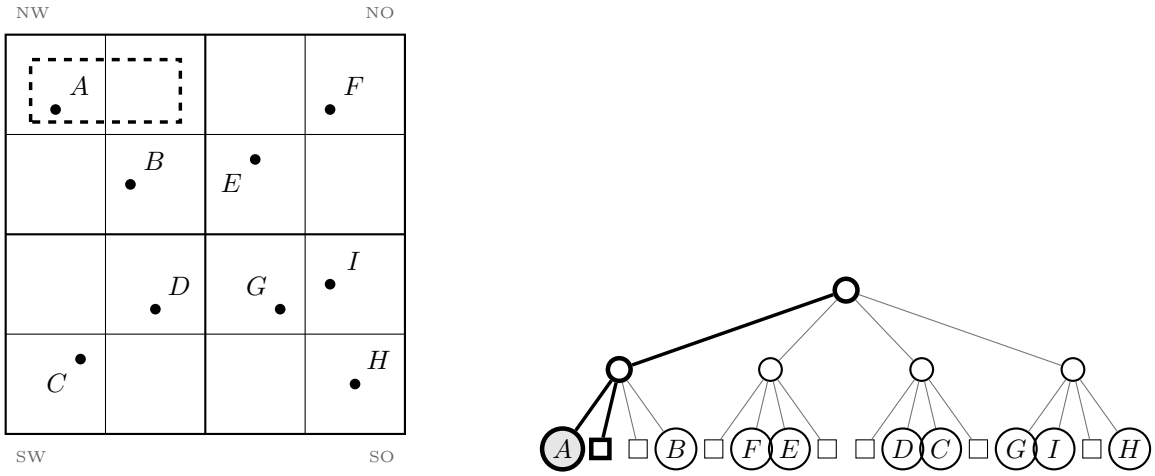
Binärbäume mit Ausgleich sind nicht das Ende der Geschichte. Drei Ausblicke, jeweils nur zum Kennenlernen.

Ausgeglichene Bäume in der Praxis

- **Rot-Schwarz-Bäume** gleichen mit einer lockereren Regel als AVL aus und brauchen dadurch weniger Rotationen. Sie stecken in den Standardbibliotheken vieler Programmiersprachen.
- **B-Bäume** haben in jedem Knoten viele Schlüssel und viele Kinder. Der Baum bleibt dadurch sehr flach. Das lohnt sich dort, wo jeder Zugriff teuer ist, zum Beispiel auf einer Festplatte: Datenbanken (Indizes) und Dateisysteme nutzen sie.

Wenn ein Knoten vier Kinder hat: der Quadtree

Ein *Quadtree* verwaltet Punkte in einer Fläche. Die Fläche wird in vier Quadranten geteilt (NW, NO, SW, SO). Enthält ein Quadrant mehr als einen Punkt, wird er wieder in vier Teile geteilt, und so weiter. Jeder Knoten hat also vier Kinder, und die Reihenfolge der Kinder ist festgelegt: NW, NO, SW, SO.



Links die geteilte Fläche mit 9 Punkten, rechts der passende Quadtree (Kinder von links nach rechts: NW, NO, SW, SO; Quadrat = leerer Quadrant). Gestrichelt: das Abfragerechteck. Dicke Knoten und Kanten: Quadranten, die das Rechteck berühren

Aufgabe: Welche Punkte liegen im gestrichelten Rechteck? Gehen Sie im Baum von der Wurzel aus vor: Welche Quadranten müssen Sie gar nicht ansehen, weil das Rechteck sie nicht berührt? (Lösung 11 am Ende des Skripts.)

Berührt ein Quadrant das Rechteck nicht, überspringt man ihn mit *einem* Vergleich samt allem, was darunter hängt. In einem Test mit 100 000 Zufallspunkten und einem Rechteck, das nur 1 % der Fläche bedeckt, genügten rund 4 400 Vergleiche (Quadranten und einzelne Punkte) statt 100 000 Punktvergleichen. Das gilt allerdings nur, wenn die Punkte einigermaßen gleichmäßig verteilt sind; ballen sie sich, wird der Baum an dieser Stelle tief, ähnlich wie der entartete Suchbaum in Abschnitt 5.

Wo Quadtrees vorkommen: bei der Kollisionsprüfung in Spielen (nur Objekte im selben Quadranten prüfen), bei Karten (beim Hineinzoomen wird jede Kachel in vier Kacheln geteilt), in Geo-

datenbanken (Abfrage „alles in diesem Kartenausschnitt“) und bei der Bildkompression (Bereiche gleicher Farbe werden zu einem Knoten zusammengefasst).

Mögliche Kurzprojekte

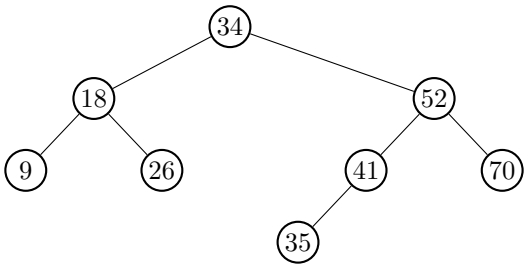
Wer Lust hat, kann eines davon vertiefen (Vorschläge, keine Pflicht):

- Einen Quadtree selbst programmieren und die Rechteckabfrage mit dem Durchprobieren aller Punkte vergleichen (Zeit messen).
- Einen B-Baum recherchieren und der Gruppe vorstellen: Wie sieht ein Knoten aus, wie wird er geteilt?
- Rot-Schwarz-Regeln und AVL-Regeln vergleichen: Wer rotiert wann?
- Bildkompression mit einem Quadtree: ein kleines Schwarz-Weiß-Bild in Quadranten zerlegen.

Weiterlesen:
Rot-Schwarz-Baum: `https://de.wikipedia.org/wiki/Rot-Schwarz-Baum`
B-Baum: `https://de.wikipedia.org/wiki/B-Baum`
Quadtree: `https://de.wikipedia.org/wiki/Quadtree`

11 Übungen

Aufgabe 1 (Begriffe). Gegeben ist der Suchbaum:



Nennen Sie Wurzel, alle Blätter, die Höhe des Baums, die Tiefe des Knotens 35 und den linken Teilbaum von 52. Wie viele innere Knoten gibt es?

Aufgabe 2 (Suchbaum). Fügen Sie die Zahlen 28, 14, 40, 7, 21, 35, 56, 18 in dieser Reihenfolge in einen leeren Suchbaum ein. Zeichnen Sie den Baum und bestimmen Sie seine Höhe. Wie hoch wird der Baum, wenn man dieselben Zahlen *sortiert* einfügt?

Aufgabe 3 (Durchläufe). Geben Sie für den Baum aus Aufgabe 2 Preorder, Inorder, Postorder und Levelorder an. Was fällt bei Inorder auf?

Aufgabe 4 (Löschen). Löschen Sie aus dem Baum von Aufgabe 2 *jeweils einzeln*: (a) 56, (b) 21, (c) 28. Nennen Sie jeweils den Fall und geben Sie den entstehenden Baum in Preorder an.

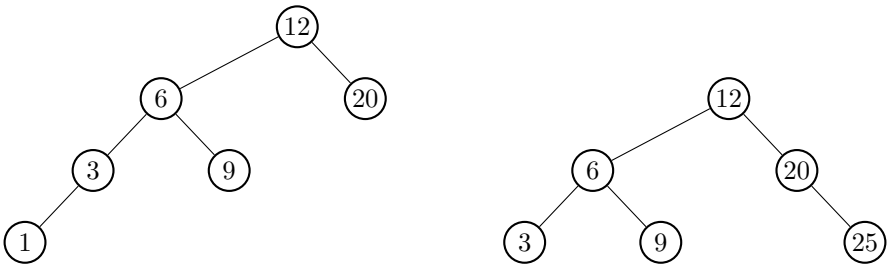
(d) *Trockenlauf:* Verfolgen Sie mit Quelltext 6 den Aufruf `loeschen(wurzel, 28)` für den Baum aus Aufgabe 2. Notieren Sie jeden Aufruf von `loeschen` und `maximum` (Wert des Knotens `k`, Wert `wert`) mit dem zurückgegebenen Knoten.

Quelltext 6: Löschen im Suchbaum (für den Trockenlauf)

```
def maximum(k):
    while k.rechts is not None:
        k = k.rechts
    return k

def loeschen(k, wert):              # liefert die (neue) Wurzel des Teilbaums
    if k is None:
        return None                 # Wert nicht vorhanden
    if wert < k.wert:
        k.links = loeschen(k.links, wert)
    elif wert > k.wert:
        k.rechts = loeschen(k.rechts, wert)
    else:                           # k ist der Knoten, der gelöscht wird
        if k.links is None:         # Blatt oder nur rechtes Kind
            return k.rechts
        if k.rechts is None:        # nur linkes Kind
            return k.links
        m = maximum(k.links)        # zwei Kinder: Maximum links
        k.wert = m.wert
        k.links = loeschen(k.links, m.wert)
    return k
```

Aufgabe 5 (Balance). Tragen Sie in beide Bäume die Balance b jedes Knotens ein. Welcher Baum ist ein AVL-Baum? Wo wird das Kriterium verletzt, und welcher Fall liegt vor?



Aufgabe 6 (Mindestknotenzahl). Wie viele Knoten hat ein AVL-Baum der Höhe 5 mindestens? Wie hoch kann ein AVL-Baum mit 100 Knoten höchstens sein?

Aufgabe 7 (AVL-Einfügen). Fügen Sie die Zahlen 38, 54, 2, 30, 4, 46, 12, 40, 14 nacheinander in einen leeren AVL-Baum ein. Geben Sie bei jeder Rotation den Knoten mit $|b| = 2$, den Fall und die Rotation an. Zeichnen Sie den Endbaum mit allen Balancen.

Aufgabe 8 (AVL-Löschen). Löschen Sie aus dem Baum von Aufgabe 7 die Zahl 2. Prüfen Sie von der Änderung aufwärts jede Balance und gleichen Sie aus.

Aufgabe 9 (Quelltext). Ein Knoten `wurzel` (Wert 10) hat das linke Kind 5 und das rechte Kind 20; 20 hat die Kinder 15 und 25. Zeichnen Sie den Baum, der nach `wurzel = lrot(wurzel)` entsteht. Welcher Wert steht dann in `wurzel.links.rechts`?

Quelltext 7: Linksrotation

```
def lrot(k):
    r = k.rechts
    k.rechts = r.links
    r.links = k
    return r
```

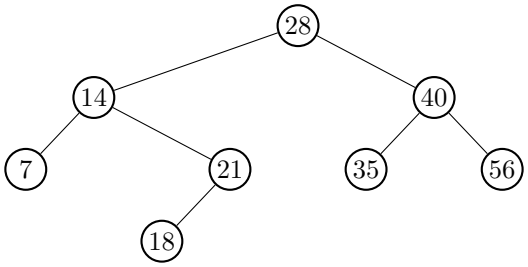
Aufgabe 10 (Reihenfolgen). Die Zahlen $1, 2, 3$ können in $3! = 6$ Reihenfolgen eingefügt werden. Wie viele verschiedene Suchbäume entstehen ohne Ausgleich? Welche Reihenfolgen führen zum selben Baum? Wie viele verschiedene Bäume entstehen mit AVL-Ausgleich?

Aufgabe 11 (Quadtree). Bearbeiten Sie die Aufgabe aus dem Ausblick (Abschnitt 10): Welche Punkte liegen im gestrichelten Rechteck? Welche Quadranten müssen Sie auf dem Weg von der Wurzel gar nicht ansehen?

12 Lösungen

Lösung 1. Wurzel 34; Blätter 9, 26, 35, 70; Höhe 3; Tiefe von 35 ist 3 (Pfad $34 \to 52 \to 41 \to 35$); der linke Teilbaum von 52 besteht aus 41 und 35 (Wurzel 41, linkes Kind 35); innere Knoten: 4 (alle außer den Blättern).

Lösung 2.



Höhe 3. Sortiert eingefügt entsteht eine Liste der Höhe 7 (jede Zahl wird rechtes Kind der vorigen).

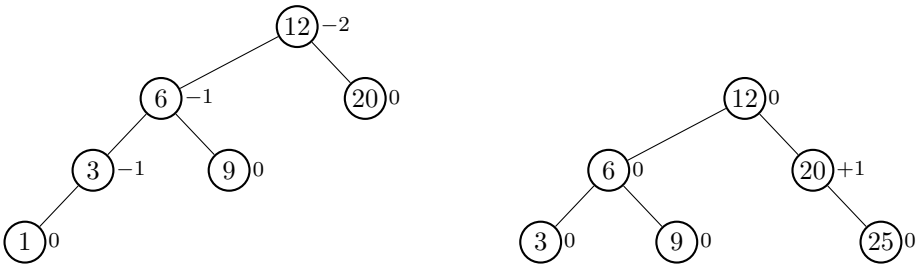
Lösung 3. Preorder: 28, 14, 7, 21, 18, 40, 35, 56. Inorder: 7, 14, 18, 21, 28, 35, 40, 56 (sortiert). Postorder: 7, 18, 21, 14, 35, 56, 40, 28. Levelorder: 28, 14, 40, 7, 21, 35, 56, 18.

Lösung 4. (a) 56 ist ein Blatt: einfach entfernen; Preorder 28, 14, 7, 21, 18, 40, 35. (b) 21 hat ein Kind (18): 18 rückt nach; Preorder 28, 14, 7, 18, 40, 35, 56. (c) 28 hat zwei Kinder: Das Maximum des linken Teilbaums ist 21 (einmal links, dann nur rechts); 21 nimmt den Platz ein, sein Kind 18 rückt an die alte Stelle von 21; Preorder 21, 14, 7, 18, 40, 35, 56.

(d) *Trockenlauf* (erste Zahl: Wert des Knotens `k`):

Aufruf	Rückgabe (Knoten)	Was passiert
`loeschen(28, 28)`	21	Wert gefunden, zwei Kinder: Maximum des linken Teilbaums holen, 28 durch 21 ersetzen, 21 im linken Teilbaum löschen
`maximum(14)`	21	von 14 nur nach rechts: 21 hat kein rechtes Kind
`loeschen(14, 21)`	14	$21 > 14$: im rechten Teilbaum weiter; das Ergebnis wird neues rechtes Kind
`loeschen(21, 21)`	18	Wert gefunden, nur ein linkes Kind: dieses (18) rückt nach

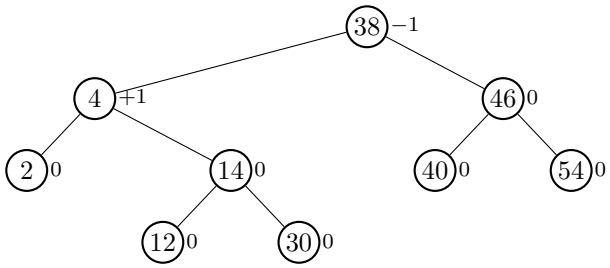
Lösung 5.



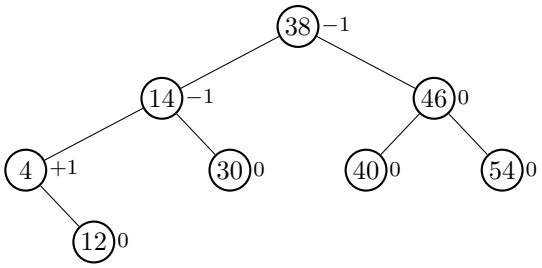
Der rechte Baum ist ein AVL-Baum. Im linken Baum ist $b(12) = -2$ (der linke Teilbaum hat Höhe 2, der rechte Höhe 0). Das linke Kind 6 hat $b = -1 \leq 0$: Fall LL, Behebung durch eine Rechtsrotation um 12.

Lösung 6. $N(2) = 4$, $N(3) = 7$, $N(4) = 12$, $N(5) = 12 + 7 + 1 = 20$. Mit $N(8) = 88 \leq 100 < N(9) = 143$ ist die Höhe höchstens 8.

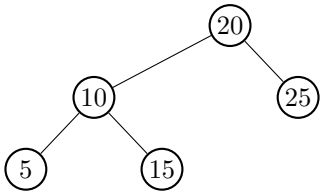
Lösung 7. Rotationen: beim Einfügen von 4: Fall RL (Rechts-Links-Doppelrotation); beim Einfügen von 40: Fall LL (Rechtsrotation); beim Einfügen von 14: Fall LR (Links-Rechts-Doppelrotation).



Lösung 8. Nach dem Löschen (wie im Suchbaum) – 2 ist ein Blatt und wird entfernt. Danach: Fall RR (Linksrotation).



Lösung 9. `r = k.rechts` ist der Knoten 20; `k.rechts = r.links` hängt 15 als rechtes Kind unter 10; `r.links = k` macht 10 zum linken Kind von 20. Neue Wurzel ist 20:



In `wurzel.links.rechts` steht 15.

Lösung 10. Ohne Ausgleich entstehen 5 verschiedene Bäume; nur die Reihenfolgen `213, 231` ergeben denselben Baum (Wurzel 2). Mit AVL-Ausgleich entsteht in allen sechs Fällen derselbe Baum: 2 als Wurzel mit den Kindern 1 und 3.

Lösung 11. Im Rechteck liegt nur der Punkt A. Das Rechteck berührt von den vier Quadranten der Wurzel nur den nordwestlichen (NW); NO, SW und SO mit zusammen 7 Punkten werden mit je einem Vergleich übersprungen. Im NW-Quadranten berühren das Rechteck nur noch die beiden oberen Teile (NW und NO); der Teil NW enthält A, der Teil NO ist leer. Der Punkt B im Teil SO wird gar nicht angesehen. Es wurden 9 Quadranten und 1 Punkt geprüft, statt alle 9 Punkte.