

# Algorithmen und Datenstrukturen SoSe25

## -Assignment 3-

Moritz Ruge

Matrikelnummer: 5600961

Lennard Wittenberg

Matrikelnummer: —

Mai 2025

## Problem 1: AVL-Bäume

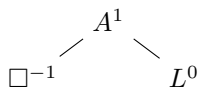
a) Fügen Sie die Schlüssel A, L, G, O, D, T, S, X, Y, Z in dieser Reihenfolge in einen anfangs leeren AVL-Baum ein. Löschen Sie sodann die Schlüssel Z, A, L. Zeichnen Sie den Baum nach jedem Einfüge- und Löschvorgang, und zeigen Sie die Rotationen, welche durchgeführt werden. Annotieren Sie dabei auch die Knoten mit ihrer jeweiligen Höhe.

⇒ Bei den Knoten die hochgestellte Zahl ist die Höhe des jeweiligen Knotens.

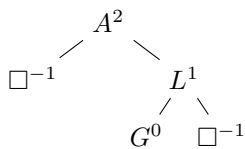
1. Einfügen: A

$A^0$

2. Einfügen: L

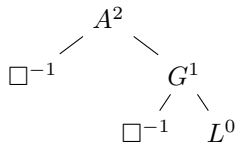


3. Einfügen: G

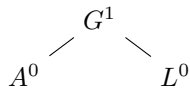


- BF-Faktor bei Knoten A ist größer als 1 ⇒ Um-balancieren der Knoten A, L, G

⇒ Rechts-Rotation der Knoten L&G

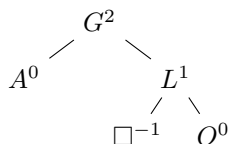


⇒ Links-Rotation der Knoten A&G

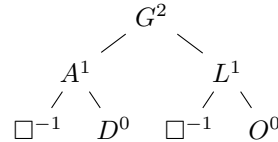


⇒ AVL-Baum ist ausgeglichen

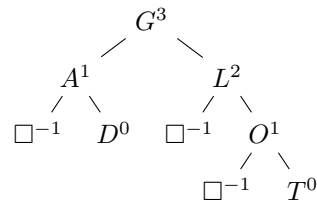
4. Einfügen: O



5. Einfügen: D

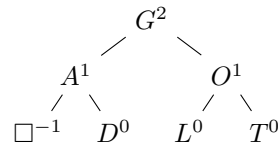


6. Einfügen: T



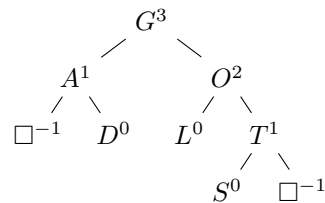
- BF-Faktor bei Knoten L ist größer als 1 ⇒ Um-balancieren der Knoten L, O, T

⇒ Rechts-Rotation der Knoten L&O

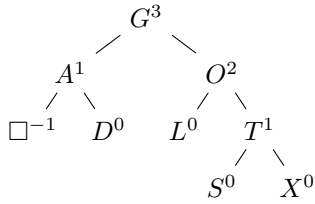


⇒ AVL-Baum ist ausgeglichen

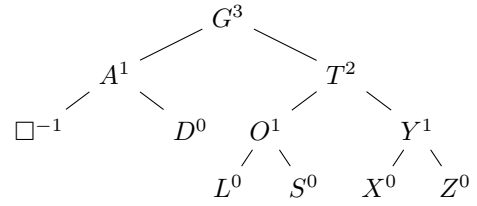
7. Einfügen: S



8. Einfügen: X

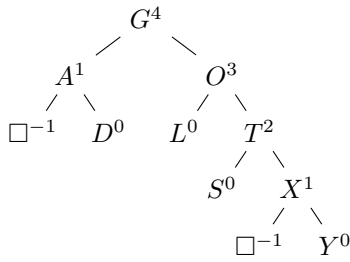


⇒ Links-Rotation der Knoten X&Y



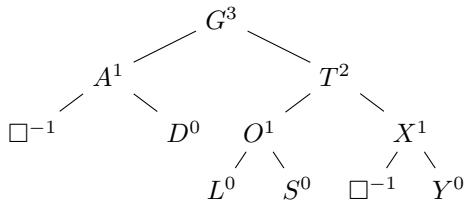
⇒ AVL-Baum ist ausgeglichen

9. Einfügen: Y



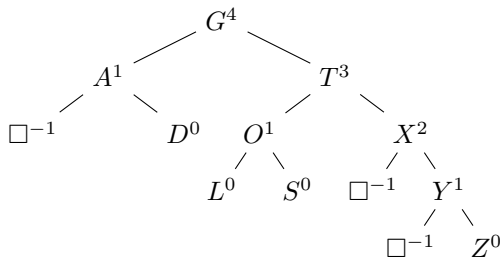
- BF-Faktor bei Knoten O ist größer als 1 ⇒ Um-balancieren der Knoten O&T

⇒ Links-Rotation der Knoten O&T



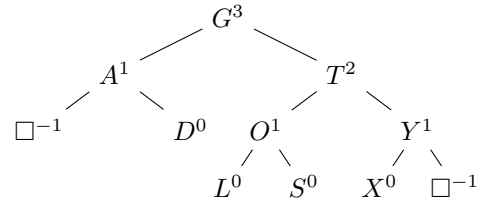
⇒ AVL-Baum ist ausgeglichen

10. Einfügen: Z

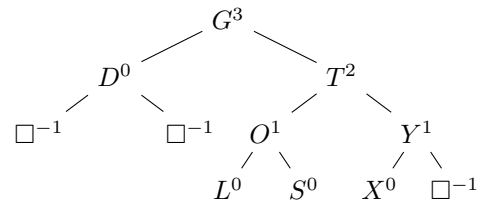


- BF-Faktor bei Knoten X ist größer als 1 ⇒ Um-balancieren der Knoten Y&Z

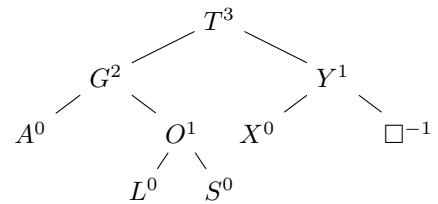
11. Lösche: Z



12. Lösche: A

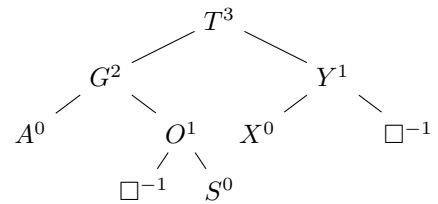


- BF-Faktor bei Knoten G ist größer als 1 ⇒ Um-balancieren der Knoten G&T



⇒ AVL-Baum ist ausgeglichen

13. Lösche: L



b) Beweisen Sie: Beim Einfügen in einen AVL-Baum wird höchstens eine (Einfach- oder Doppel-)Rotation ausgeführt. Gilt das auch beim Löschen (Begründung)?

**Annahme:** Beim Einfügen in einen AVL-Baum wird höchstens eine (Einfach- oder Doppel-)Rotation ausgeführt.

- In einem AVL-Baum gelten die Eigenschaften eines BTS.
- In einem AVL-Baum muss gelten: Die Differenz zwischen der Höhe des Linken Teilbaum und der Höhe des Rechten Teilbaum darf maximal 1 sein.

base-case AVL-Baum mit einem Knoten:

$r \rightarrow h: 0$

insertion (n):

```

'''
      r  h: 0
     /  \
    n    nil  hl: 1, hr: 0
'''

```

$|hl - hr| \leq 1$  true  $\rightarrow$  keine Rotation nötig!

**Annahme gilt im base-case**

**I.S:** Wenn die Annahme bei  $n$  Insertions gilt, gilt sie auch  $n+1$  Insertions. Wenn nach  $n$  insertions ein AVL-Baum die AVL-Eigenschaften weiterhin erfüllt kann es zu mehreren Fällen bei einer  $n + 1$  insertion kommen:

Fallunterscheidung:

1. Nach einer Insertion wird kein Knoten unbalanciert.  $\rightarrow$  keine Rotation notwendig
2. Nach einer Insertion wird ein Knoten unbalanciert, da einer der Teilbäume tiefer ist als der andere und  $|hl - hr| \leq 1$  nicht mehr gilt. Da die Unbalance nur nach einer einzelnen Insertion auftreten kann, ist der Höhen Unterschied immer 2.

2.1. der Linke Teilbaum eines Knotens ist Tiefer als der Rechte Teilbaum.

''' Beispielabschnitt für links Unbalance irgendwo in einem AVL-Baum

```

      n
     /  \
    n1   n1
   /
  n2
 /
n3
'''

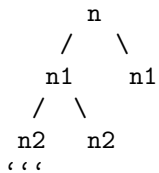
```

Am tiefsten unbalanciertem Knoten (pivot) wird eine rechts-Rotation ausgeführt.

- Die Rotation wird am tiefsten unbalanciertem Knoten ausgeführt, weil nach einer Rotation die Höhe des betroffenen Teilbaums wieder den selben Wert hat, wie vor der  $n+1$  Insertion.

- Der Elternknoten des pivot-Knotens wird zum neuen Rechten Kind des pivot-Knotens.
- Das alte Rechte Kind des pivot-Knotens wird zum neuen linken Kind des neuen Rechten Kindknotens.

“ “



“ “

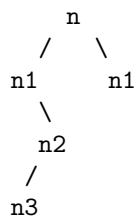
2.2 Der Rechte Teilbaum eines Knotens ist Tiefer als der Linke Teilbaum. Zeichnungen sind analog. Am tiefsten unbalanciertem Knoten (pivot) wird eine links-Rotation ausgeführt.

- Die Rotation wird am tiefsten unbalanciertem Knoten ausgeführt, weil nach einer Rotation die Höhe des betroffenen Teilbaums wieder den selben Wert hat, wie vor der  $n+1$  Insertion.
- Der Elternknoten des pivot-Knotens wird zum neuen Linken Kind des pivot-Knotens.
- Das alte Linke Kind des pivot-Knotens wird zum neuen Rechten Kind des neuen Linken Kindknotens.

2.3 Der Rechte-Linke Teilbaum eines Knotens ist Tiefer als der Rechte Teilbaum. Es wird erst eine links-Rotation am tiefsten unbalanciertem Knoten (als Pivot Elternknoten) durchgeführt und dann eine weitere rechts-Rotation mit dem selben pivot-Knoten durchgeführt.

- Die Rotationen selbst funktionieren genau so wie in 2.1 und 2.2 beschrieben und sequentiell am selben pivot-Knoten.
- Auch bei Doppel-Rotationen gilt: Nach der Rotation sind die Höhenverhältnisse der Teilbaume wieder so wie vor der Insertion  $n+1$

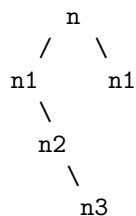
“ “ Beispielabschnitt für rechts-links Unbalance irgendwo in einem AVL-Baum



“ “

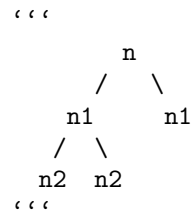
nach links-Rotation

“ “



“ “

nach rechts-Rotation



2.4 Der Linke-Rechte Teilbaum eines Knotens ist Tiefer als der Linke Teilbaum. Zeichnungen sind analog. Es wird erst eine rechtss-Rotation am tiefen unbalanciertem Knoten (als Pivot Elternknoten) durchgeführt und dann eine weitere links-Rotation mit dem selben pivot-Knoten durchgeführt.

- Die Rotationen selbst funktionieren genau so wie in 2.1 und 2.2 beschrieben und sequentiell am selben pivot-Knoten.
- Auch bei Doppel-Rotationen gilt: Nach der Rotation sind die Höhenverhältnisse der Teilbaume wieder so wie vor der Insertion  $n + 1$

Man sieht man benötigt maximal eine (Einfach- oder Doppel-) Rotation um nach einer Insertion die AVL-Eigenschaft zu erhalten.

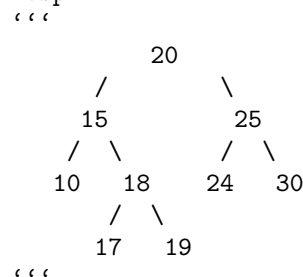
Annahme gilt für  $n + 1$

□

### Löschen:

- Angenommen man hat einen beliebigen AVL-Baum.
- Beim Löschen eines beliebigen Knotens kann es sein, dass eine komplexere Unbalance eintritt, diese ist nicht durch einmaliges rotieren lösbar.

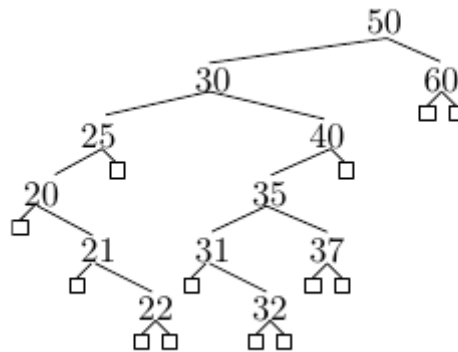
bsp:



wenn man 15 entfernen würde, wäre die entstehende Unbalance nicht mit einer einmaligen Rotation lösbar. Daher gilt die Annahme meiner Meinung nach nur für das Einfügen (Insertion)

## Problem 2: findRange

a) Betrachten Sie den folgenden binären Suchbaum: Wo befinden sich die Schlüssel, die kleiner sind als 37? Wo befinden sich die Schlüssel, die größer sind als 21? Wo befinden sich die Schlüssel, die zwischen 21 und 37 liegen?



### Knoten < 37:

- Alle Knoten, die von den 37 Knoten im Baum übrig bleiben (visuell)
- Knoten 30 und der gesamte linke Teilbaum von 30 und der linke Teilbaum von 40 (ohne 37 selbst)
- Knoten < 37 umfassen: {30, 25, 20, 21, 22, 35, 31, 32}

### Knoten > 21:

- Alle Knoten rechts vom 21. Knoten im Baum (visuell)
- Alle Vorgänger Knoten des Knoten 21, für die gilt:  $n > 21$
- Die rechten Teilbäume dieser Vorgänger
- Knoten > 21: 22, 25, 30, 40, 35, 31, 32, 37, 50, 60

### Knoten > 21 und Knoten < 37:

- Alle Knoten, die visuell links von 37 und rechts von 21 sind.
- Knoten im rechten Teilbaum von 21
- Der Knoten 30 und alle Knoten des linken Teilbaums von 30, für die vergoldeten  $n > 21$
- Knoten im linken Teilbaum von 40 außer dem Knoten 37 selbst.
- **Knoten > 21 und Knoten < 37:** 22, 25, 30, 31, 32, 35

b) Beschreiben Sie, wie man in einem AVL-Baum mit  $n$  Schlüsseln die Operation  $\text{findRange}(k_1, k_2)$  implementieren kann, die alle Schlüssel  $k$  liefert, für die  $k_1 \leq k \leq k_2$  ist. Die Laufzeit soll  $O(\log n + s)$  betragen. Dabei ist  $s$  die Anzahl der gelieferten Schlüssel.

**Idee/Annahme:**

- Ein AVL-Baum ist ein balancierter binärer Suchbaum, d.h. für jeden Knoten gilt:
- Alle Schlüssel im linken Teilbaum  $T_L$  sind kleiner als der Elternknoten
- Alle Schlüssel im rechten Teilbaum  $T_R$  sind größer als der Elternknoten
- Die allgemeine Laufzeit eines AVL-Baums beträgt  $O(\log n)$

**Herangehensweise:**

- Wenn der Schlüssel  $k_1 <$  Elternknotens ist gehen wir weiter in den linken Teilbaum  $T_L$ , bis wir bei  $k_1$  oder  $NULL$  sind.
- Wenn der Schlüssel  $k_2 >$  Elternknoten ist gehen wir in den rechten Teilbaum  $T_R$ , bis wir bei  $k_2$  oder  $NULL$  sind.
- Sollte während des Vergleichs der Knoten:  $k_1 \leq \text{Knoten} \leq k_2 \rightarrow$  speichern wir den Wert
- So laufen wir durch den gesamten Baum, bzw. durch alle Knoten die sich innerhalb des Intervalls von  $[k_1, k_2]$  befinden.
- Dabei nutzen wir die Eigenschaften eines AVL-Baums aus (das er geordnet ist) und betreten beim Suchen nur die relevanten Teilbäume.

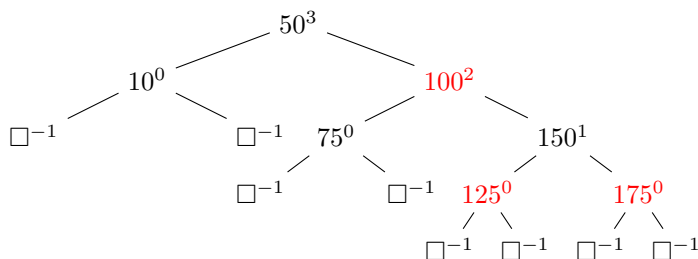


## Problem 3: Rot-Schwarz Bäume

Ein rot-schwarz Baum ist ein binärer Suchbaum, den wir auf die folgende Weise erweitern: Jeder Knoten und jeder leere Teilbaum erhält eine Farbe (rot oder schwarz), so dass die folgenden Regeln gelten: (i) die Wurzel ist schwarz; (ii) die leeren Teilbäume sind schwarz; (iii) die Kinder eines roten Knoten sind schwarz; und (iv) die schwarze Tiefe aller leeren Teilbäume ist gleich, d.h., für alle leeren Teilbäume ist die Anzahl der schwarzen Knoten auf dem Pfad von der Wurzel zum jeweiligen Teilbaum gleich.

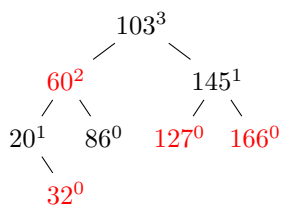
a) Zeichnen Sie drei Beispiele für rot-schwarz Bäume und erklären Sie, warum diese jeweils die Regeln für einen rot-schwarz Baum erfüllen.

### 1. Beispiel



- (i) Wurzel ist schwarz
- (ii) alle leeren Teilbäume sind schwarz
- (iii) Kinderknoten von 100 sind schwarz(75,150)
- (iv) :
  - linker Teilbaum(10): von der Wurzel bis zu den leeren Teilbäumen haben wir 1 schwarzen Knoten (10)
  - rechter Teilbaum(100-150-175): von der Wurzel bis zu den leeren Teilbaum haben wir 1 schwarzen Knoten (150)

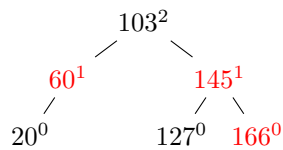
### 2. Beispiel



- (i) Wurzel ist schwarz
- (ii) alle leeren Teilbäume sind schwarz (aus Platzgründen weggelassen)
- (iii) Kinderknoten von 60 sind schwarz(20,86)
- (iv) :

- linker Teilbaum(60-20-32): von der Wurzel bis zu den leeren Teilbäumen haben wir 1 schwarzen Knoten (20)
- linker Teilbaum(60-86): von der Wurzel bis zu den leeren Teilbäumen haben wir 1 schwarzen Knoten (86)
- rechter Teilbaum(145-127): von der Wurzel bis zu den leeren Teilbaum haben wir 1 schwarzen Knoten (145)
- rechter Teilbaum(145-166): von der Wurzel bis zu den leeren Teilbaum haben wir 1 schwarzen Knoten (145)

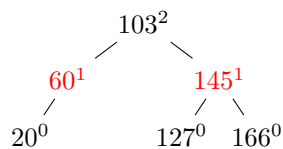
### 3. Beispiel (Falsch)



- (i) Wurzel ist schwarz
- (ii) alle leeren Teilbäume sind schwarz (aus Platzgründen weggelassen)
- (iii) Kinderknoten von (60,145) ist schwarz(20,127)
- (iii) Kinderknoten von 145 ist rot(166) - Verletzung der Schwarz-Rot-Baum-Eigenschaft!

⇒ Die ist aufgrund der Verletzung von (iii) kein gültiger Schwarz-Rot-Baum! Siehe **3.1** für die Korrigierte Variante.

#### 3.1 Beispiel (Richtig)



- (i) Wurzel ist schwarz
- (ii) alle leeren Teilbäume sind schwarz (aus Platzgründen weggelassen)
- (iii) Kinderknoten von (60,145) ist schwarz(20,127,166)
- (iv) :
  - linker Teilbaum(60-20): von der Wurzel bis zu den leeren Teilbäumen haben wir 1 schwarzen Knoten (20)
  - rechter Teilbaum(145-127): von der Wurzel bis zu den leeren Teilbaum haben wir 1 schwarzen Knoten (127)
  - rechter Teilbaum(145-166): von der Wurzel bis zu den leeren Teilbaum haben wir 1 schwarzen Knoten (166)

b) Sei  $T$  ein rot-schwarz Baum, und sei  $s$  die schwarze Tiefe der leeren Teilbäume. Zeigen Sie, dass  $T$  mindestens  $2s - 1 - 1$  schwarze Knoten besitzt. Was folgt daraus über die Mindestanzahl von Knoten in einem rot-schwarz Baum mit Höhe  $h$ ? Folgern Sie: ein rot-schwarz Baum mit  $n$  Knoten hat Höhe  $O(\log n)$ .

**Annahme:** Anzahl der Schwarzen Knoten in einem red-black-tree:  $2^{s-1} - 1$ , dabei gelten (i)-(iv) aus der Aufgabenstellung.

**base-case:** red-black-tree,  $s = 2$  (leerer Baum mit Wurzel):

```

  r      | depth: 1 (root is black)
 / \
bn bn   | depth: 2 (black null leaves)

```

nach (i) gilt:  $Num_{blackNodes} = 1$   
 $Num_{blackNodes} = 2^{2-1} - 1 = 2^0 - 1 = 2 - 1 = 1$

Die Annahme gilt im Base-case, Voraussetzung: die Formel  $2^{s-1} - 1$  gilt für red-black-trees mit  $s = k$  schwarzer Tiefe.

**I.S** red-black-tree, mit  $s = k + 1$  :  
nach (i)-(iv) gilt:

- Die Wurzel ist Schwarz
- Jeder Teilbaum der Wurzel hat eine schwarze Tiefe von  $k$
- Jeder Teilbaum hat nach Annahme mindestens  $2^{k-1} - 1$  schwarze Knoten

$Num_{blackNodes} \geq 1 + 2(2^{k-1} - 1) = 1 + 2^1 * 2^{k-1} - 2 = 2^k - 1 \mid k = s - 1$   
 $Num_{blackNode} \geq 2^{s-1} - 1$

Die Formel aus der Annahme gilt auch im Induktionsschritt.

Laut Definition ist die schwarze Tiefe die Anzahl der Schwarzen Knoten von der Wurzel, bis einem Blatt, wobei Blätter ebenfalls schwarz sind.

d.h für die Anzahl schwarzer Knoten im einem red-black-tree gilt:  $Num_{blackNode} \geq h/2$ , wobei  $h$  die Höhe des gesamten Baumes ist.

daraus folgt:  $Num_{blackNodes} \leq 2^{h/2-1} - 1$  — nach  $h$  umformen:

$h \geq 2(\log(Num_{blackNodes} + 1) + 1) \rightarrow h \geq 2(\log(n + 1) + 1)$  die Dominate Komplexitätsklasse ist  $O((\log n))$

daher gilt für die Höhe eines red-black-tree  $h \leq O(\log(n))$

Quellen:

<https://medium.com/data-science/understanding-time-complexity-with-python-examples-2bda6e8158a7>  
<https://www.geeksforgeeks.org/introduction-to-red-black-tree/#interesting-points-about-redblack-t>