Datenstrukturen & Algorithmen 18. März, 2015

Handout 4

Sebastian Millius, Sandro Feuz, Daniel Graf

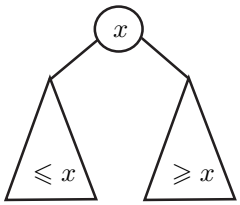
Thema: Binäre Suchbäume, AVL Bäume

Links

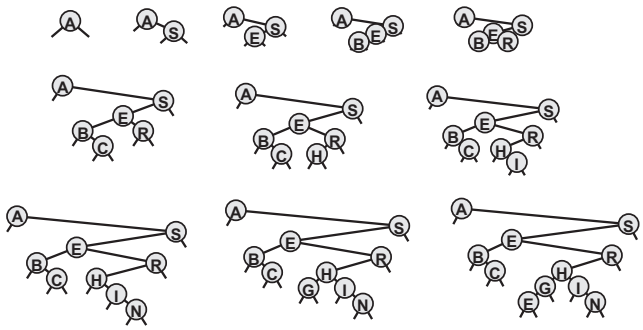
- Ein gutes Visualisierungs-Applet für AVL-Bäume:
  http://www.cs.jhu.edu/~goodrich/dsa/trees/avltree.html
  (http://goo.gl/n0Bb)
- Slides von Michael Belfrage:
  http://www.belfrage.net/eth/d&a/pdf/uebung4_h.pdf
  (http://goo.gl/HqQwM)
- MIT OpenCourseWare: AVL Implementation in Python:
  http://ocw.mit.edu/ans7870/6/6.006/s08/lecturenotes/search.htm
  (http://goo.gl/UR1hx)

Binäre Suchbäume

Ein *Binärer Suchbaum* ist ein binärer Baum, der dem Suchbaumkriterium genügt: Für jeden Knoten x gilt: Die Schlüssel im linken Teilbaum von x sind allesamt kleiner als der Schlüssel von x, und dieser ist wiederum kleiner als sämtliche Schlüssel im rechten Teilbaum von x.



Einfüge-Beispiel:



Operationen

- **Suche**: Die Suche kann iterativ formuliert werden

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ key[x]
2        if k < key[x]
3              x = left[x]
4        else x = right[x]
5  return x
```

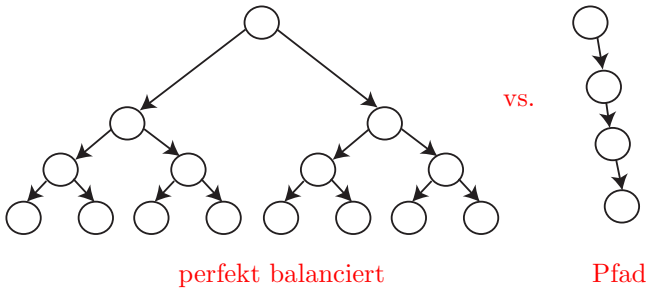
- **Minimum**: Das Minimum (das "linkeste" Element) in einem binären Suchbaum, kann gefunden werden, indem man von der Wurzel immer dem linken Kind folgt

```
TREE-MINIMUM(x)
1  while left[x] ≠NIL
2        x = left[x]
3  return x
```

- **Maximum**: analog

Stop and Think: Operationen in Binären Suchbäumen Der Nachfolger eines Schlüssels x in einem binären Suchbaum ist der kleinste Schlüssel der grösser als x ist. Wie kann der Nachfolger effizient gefunden werden?

Die Komplexität der Operationen *Einfügen*, *Suchen* ... ist $\mathcal{O}(h)$, wobei h die Höhe des Baumes ist. Bei einem binären Suchbaum ist h zwischen $\log n$ und n, d.h. im Worst Case verhält sich ein binärer Suchbaum nicht besser als eine Linked List.



Balancierte Suchbäume garantieren eine Baumhöhe $h = \mathcal{O}(\log n) \rightsquigarrow$ Operationen in $\mathcal{O}(\log n)$

Durchlaufordnungen

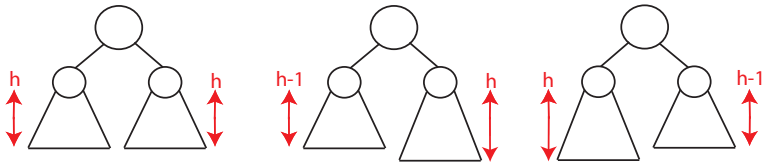
Eine Durchlaufordnung ist eine Art, wie ein Baum traversiert (durchlaufen) werden kann. Folgend die vier am häufigsten verwendeten Traversierungsmöglichkeiten, die - beginnend von der Wurzel - rekursiv ausgeführt werden können:

- **Pre**order: **Wurzel**, Linker Teilbaum, Rechter Teilbaum
  Dies entspricht der sogenannten Tiefensuche (Depth-First-Traversal)
  Think about it: Einen binären Suchbaum mit einer gegebenen Preorder Seqeunz erhält man, in dem die Elemente der Sequenz in einen anfangs leeren binären Suchbaum einfügt.
- **Post**order: Linker Teilbaum, Rechter Teilbaum, **Wurzel**
  Think about it: Einen binären Suchbaum mit einer gegebenen Postorder Seqeunz erhält man, in dem die Elemente der Sequenz rückwärts (d.h. von hinten beginnend) in einen anfangs leeren binären Suchbaum einfgt.
- **In**order: Linker Teilbaum, **Wurzel**, Rechter Teilbaum
  Die Knoten werden auf den Boden projeziert. Dies ergibt für binäre Suchbäume gerade die sortierte Liste
- Levelorder: Bei dieser Traversierung gibt man jeden Knoten eines Levels aus (von links nach rechts), bevor man zum nächst tieferen Level geht. Dieses Vorgehen ist weitgehend als *Breitensuche* (Breadth-First-Search) bekannt.

AVL Bäume

AVL Bäume sind balancierte binäre Suchbäume. Sie sind benannt nach G.M. Adel'son-Vel'skii und E.M. Landis.

AVL Baum-Strukturbedingung: die Höhen des linken und des rechten Teilbaums jedes Knotens unterscheiden sich höchstens um ± 1.



Durch den Erhalt dieser Strukturbedingung wird erreicht, dass die Höhe des Baumes jeweils $\mathcal{O}(\log(n))$ ist und beispielsweise nicht zu einer Liste degenerieren kann. Die Ausbalanciertheit erreicht man mittels *Rotationen*. Diese Rotationen sind im Buch *Algorithmen und Datenstrukturen*, S. 271ff genau beschrieben. Auch hier empfiehlt sich einmal mehr, sich mittels Visualisierungstool mit dem AVL-Baum vertraut zu machen!

Die Suche in einem AVL Baum erfolgt wie in einem natürlichen binären Suchbaum. Beim Einfügen und Entfernen von Knoten müssen evtl. Rotationen angewendet werden um die Ausbalanciertheit zu garantieren.

- **Einfügen**: Einfügen wie in binärem Suchbaum, danach solange von unten nach oben rotieren (mittels Einfach- und Doppelrotationen), bis der Baum wieder balanciert ist (höchstens eine Rotation (einfach oder doppel) ist notwendig).

  Es sei x der Knoten, bei dem die AVL Bedingung verletzt ist. x ist *nicht* der Knoten, den wir eingefügt haben. Es können folgende Fälle auftreten: der neue Knoten wurde eingefügt im

  1. linken Teilbaum des linken Kindes von x
  2. rechten Teilbaum des linken Kindes von x
  3. linken Teilbaum des rechten Kindes von x
  4. rechten Teilbaum des rechten Kindes von x

  Die Fälle 1 & 4 werden gelöst mit einer *Einfachrotation*. Die Fälle 2 & 3 mit einer *Doppelrotation*.

- **Löschen eines Knotens**

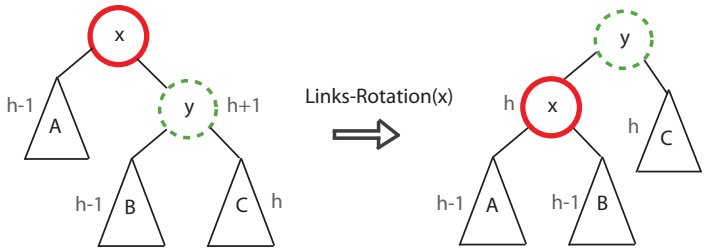
  - Suche nach dem Knoten
  - Falls gefunden $\rightsquigarrow$ Ersetzen mit symmetrischem Nachfolger/Vorgäger (Blatt!)
  - Bei dem Blatt beginnend nach oben: Rebalancierung

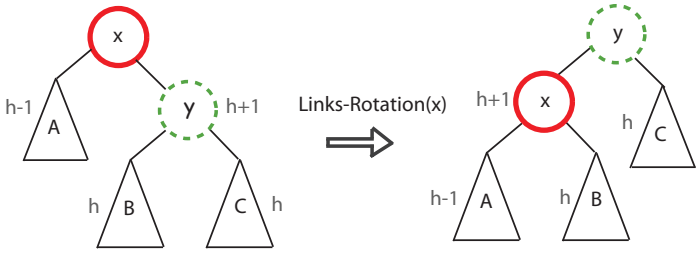
  Es können mehrere Rotationen notwendig sein, damit diese Operation abgeschlossen und die Strukturbedingung wieder erfüllt ist.

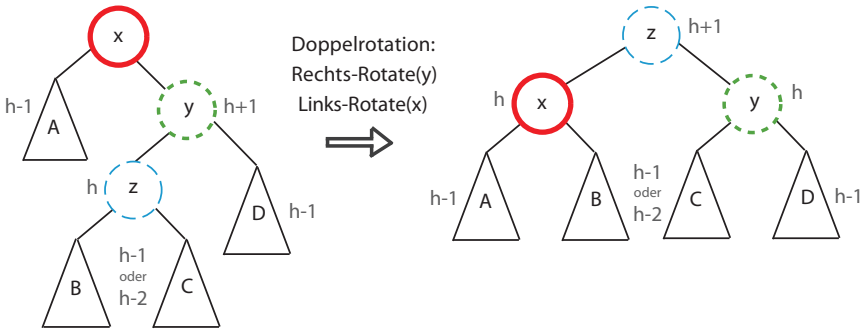
Stop and Think: Are you AVL?
Gebe einen Algorithmus an, der für einen binären Baum entscheidet ob dieser ein AVL Baum ist.

Rotationen

Um die AVL-Aufgaben von Hand zu lösen, verfährt man am besten, wie Michael Belfrage dies auf seinen Übungsslides von 2008 darstellt.
http://www.belfrage.net/eth/d&a/pdf/uebung4_h.pdf (http://goo.gl/HqQwM)

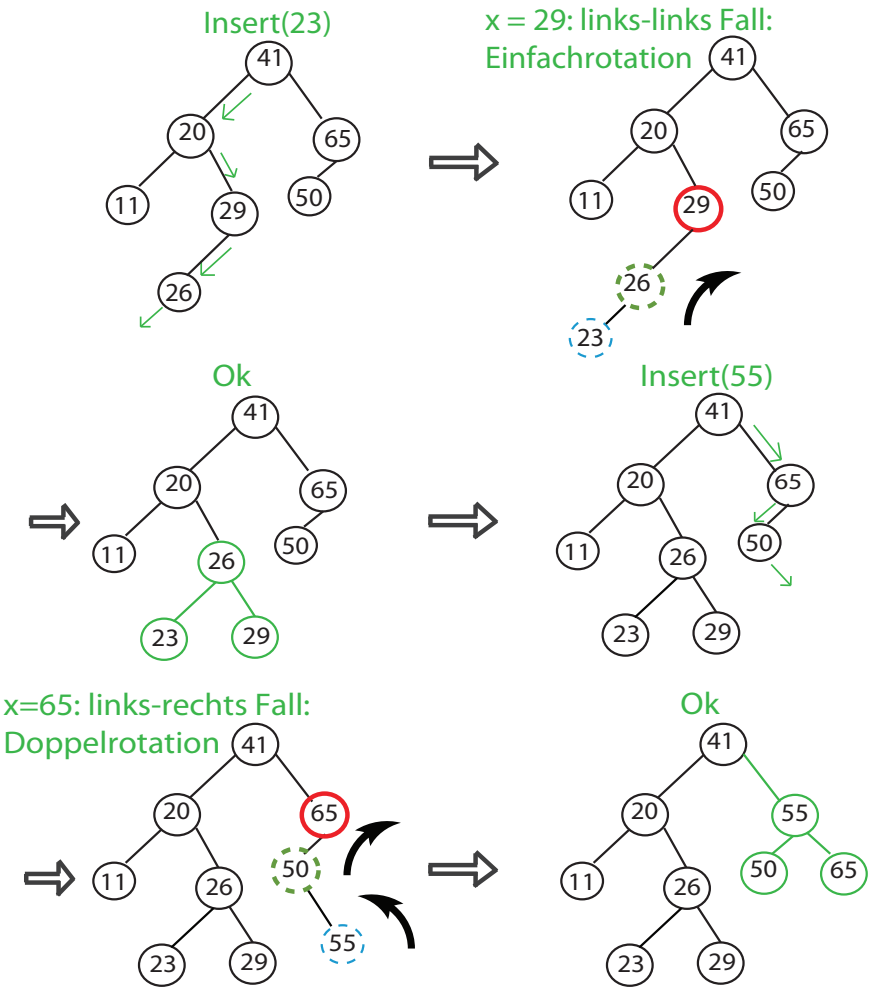






1. die unbalancierten Knoten rot färben (hier x)
2. dessen schwereres Kind grün färben (hier y)
3. Fallunterscheidung
   - grün's Kinder sind gleich schwer $\rightsquigarrow$ *Einfachrotation*
   - sonst grün's schwereres Kind blau färben (hier z)
     falls rot $<$ grün $<$ blau $\rightsquigarrow$ Einfachrotation
     sonst $\rightsquigarrow$ Doppelrotation

Beispiel:



Stop and Think: Balancierte Suchbäume *Problem*: Lesen von n Zahlen und Anzeigen in sortierter Reihenfolge. Zur Verfügung steht ein balancierter Suchbaum der die Operationen *search, insert, delete, minimum, maximum, successor, predecessor* in $O(\log n)$ ergmöglicht.

1. Wie kann man in $\mathcal{O}(n \log n)$ sortieren mit nur *insert* und *in-order* Traversierung?
2. Wie kann man in $\mathcal{O}(n \log n)$ sortieren mit nur *minimum, successor, insert*?
3. Wie kann man in $\mathcal{O}(n \log n)$ sortieren mit nur *minimum, insert, delete*?

Aufgabe

Fügen Sie in einen anfangs leeren AVL-Baum folgende Schlüssel der Reihe nach ein:

$$8, 4, 5, 12, 16, 19, 18, 3, 1, 0$$

Zeichnen Sie dabei den Baum nach jeder Einfügeoperation sowie nach jeder Rotation und Doppelrotation. Löschen Sie danach im entstandenen Baum die Schlüssel $18, 3, 4, 5, 8$ der Reihe nach. Benutzen Sie falls benötigt den symmetrischen Nachfolger. Zeichnen Sie den Baum nach jeder Löschoperation und nach jeder Rotation.