Wörterbücher

Dynamische Menge an Elementen, die über Schlüssel definiert sind:

Paare(Schlüssel, Wert)

Operationen:
element search*(schlüssel* x*)*
void insert*(element* e*)*
void delete(*schlüssel* x)
element max/min()
void traverse()

Realisirung:

- Verketette Liste
- Feld
- BitVector mit Kapazität m

	Search	**Insert**	**Delete**	**Max/Min**	**Traverse**
Liste	O(n)	O(n)	O(n)	O(n)	$O(n^2)$
Feld	O(log n)	O(n)	O(n)	O(1)	O(n)
BitVector	O(1)	O(1)	O(1)	O(m)	O(m)

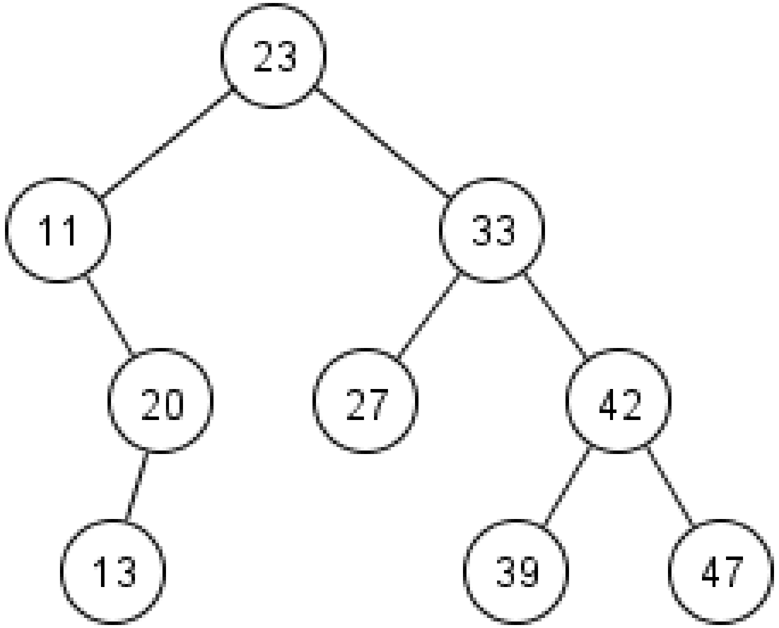
Binäre Suchbäume

Ein binärer Suchbaum ist ein binärer Baum mit der Folgender Eigenschaft:

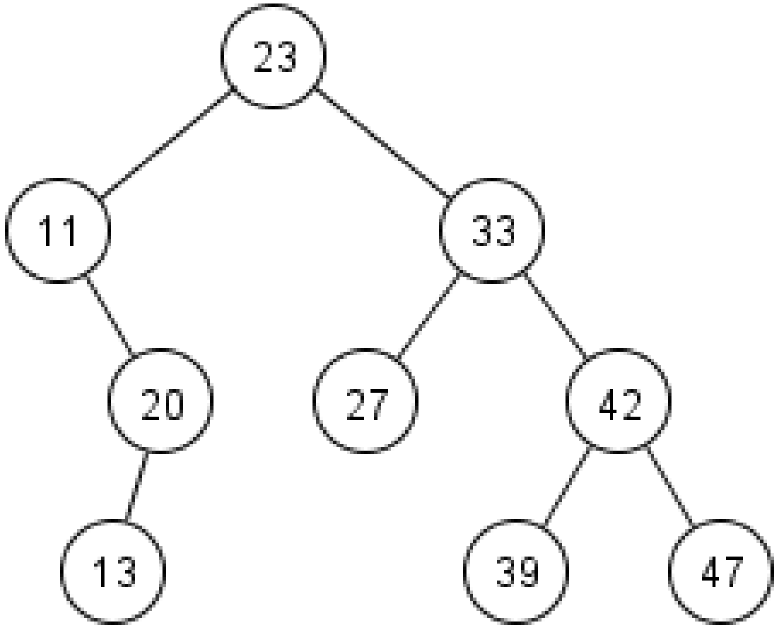
- z.key > x.key ∀z im rechten Teilbaum von x
- z.key < x.kez ∀z im linken Teilbaum von x

Knoten *x*:
knoten x.left
knoten x.right
knoten x.parent
schlüssel x.key
wert x.value

A={ 23 , 33 , 27 , 11 , 42 , 20 , 13 , 39 , 47 }



Binäre Suchbäume




```
void traverse (Knoten  x)
if x != NULL then
    traverse(x.left)
    print(x)
    traverse(x.right)
```

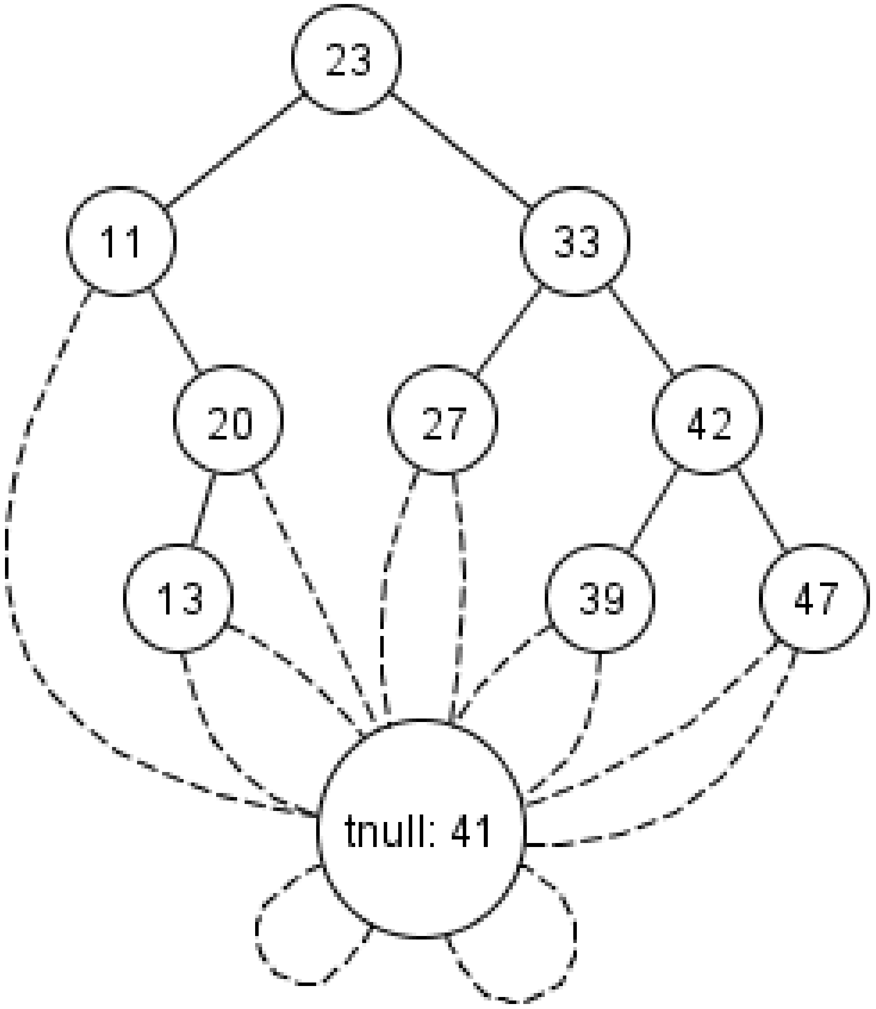
```
Knoten max (Knoten x)
if x.right != NULL
    then return Max(x.right)
    else return x
```

Binäre Suchbäume

Suchen nach 41

Wächter (Sentinel) TNULL

- Speziller Knoten, der kein Element permanent abspeichert
- Alle Null Zeiger zeigen jetzt zum Sentinel
- Sentinel ist dereferenzierbar und erlaubt, Speziallfälle zu vermeiden
- TNULL.left = TNULL
- TNULL.right = TNULL



Binäre Suchbäume

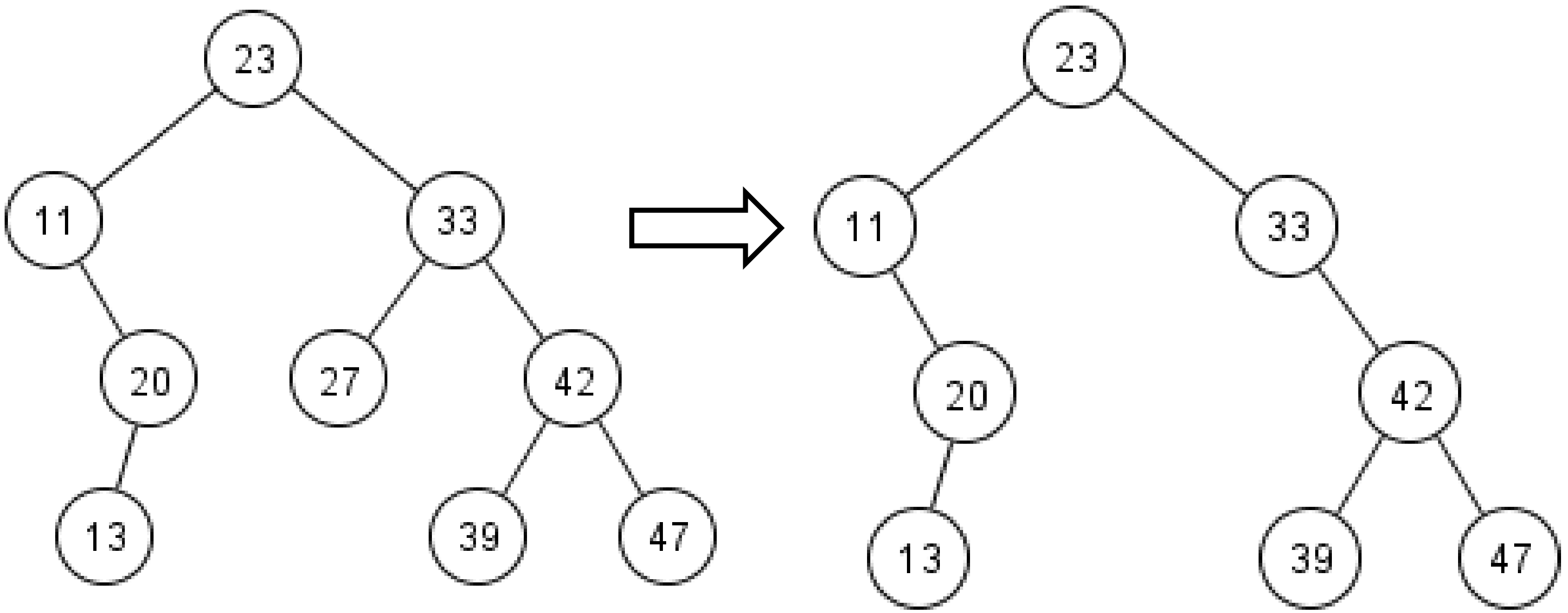
```
Knoten search (Knoten  x, Schlüssel k)
TNULL.key = k
z = sentinel_search(x, k)
if z != NULL
    then return z
    else return NULL

Knoten sentinel_search (Knoten  x, Schlüssel k)
if k < x.key  then return sentinel_search(x.left, k)
else if k > x.key then return sentinel_search(x.right, k)
else return x
```

```
void insert (Knoten  x, Schlüssel k)
if x=TNULL then x = makenode( k )
else if k < x.key  then insert(x.left, k)
else if k > x.key  then insert(x.right, k)
else key k already in tree
```

Hängt neues Blatt mit Schlüssel *k* an die richtige Stelle im Baum.

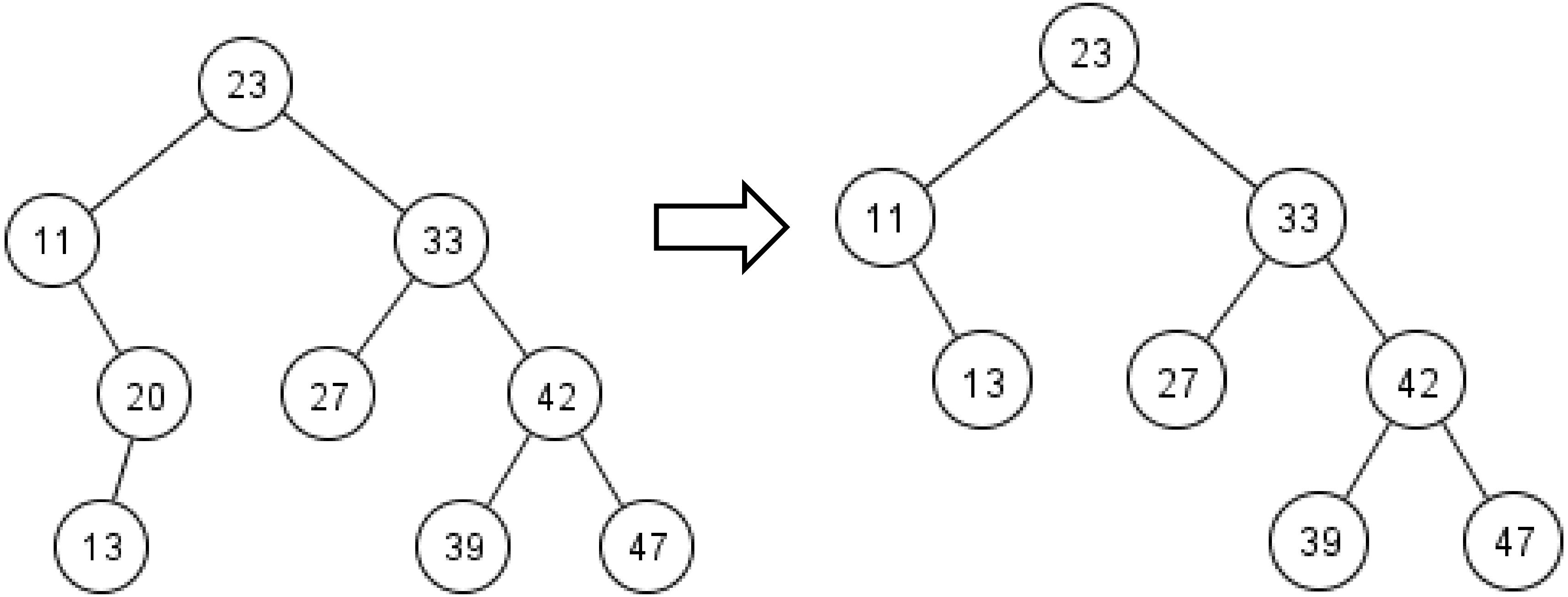
Binäre Suchbäume

27 löschen:



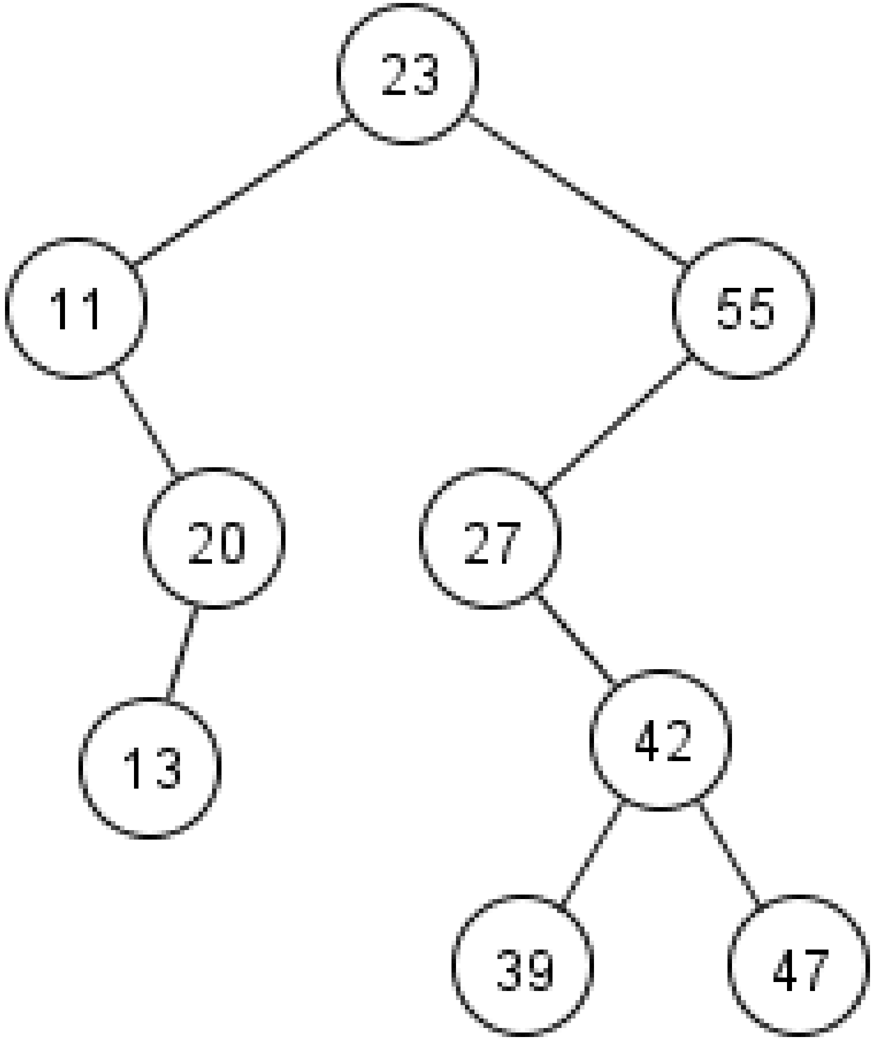
Binäre Suchbäume

20 löschen:



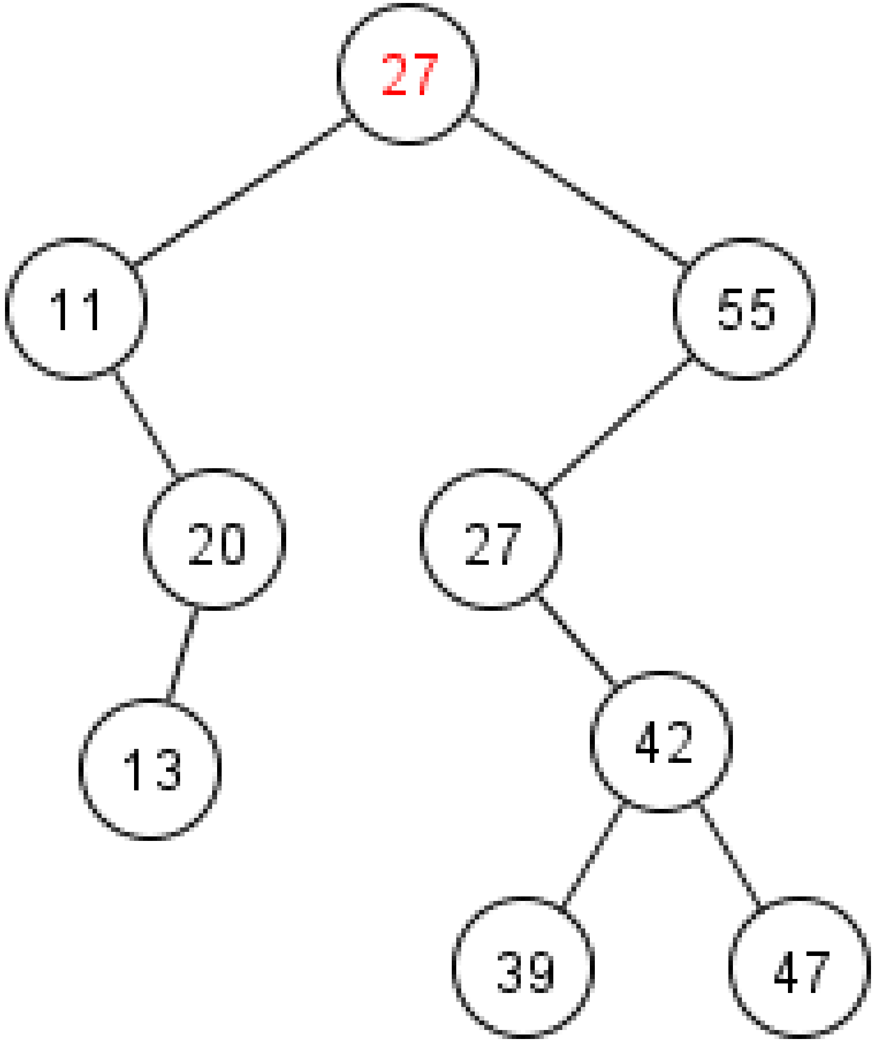
Binäre Suchbäume

23 löschen:



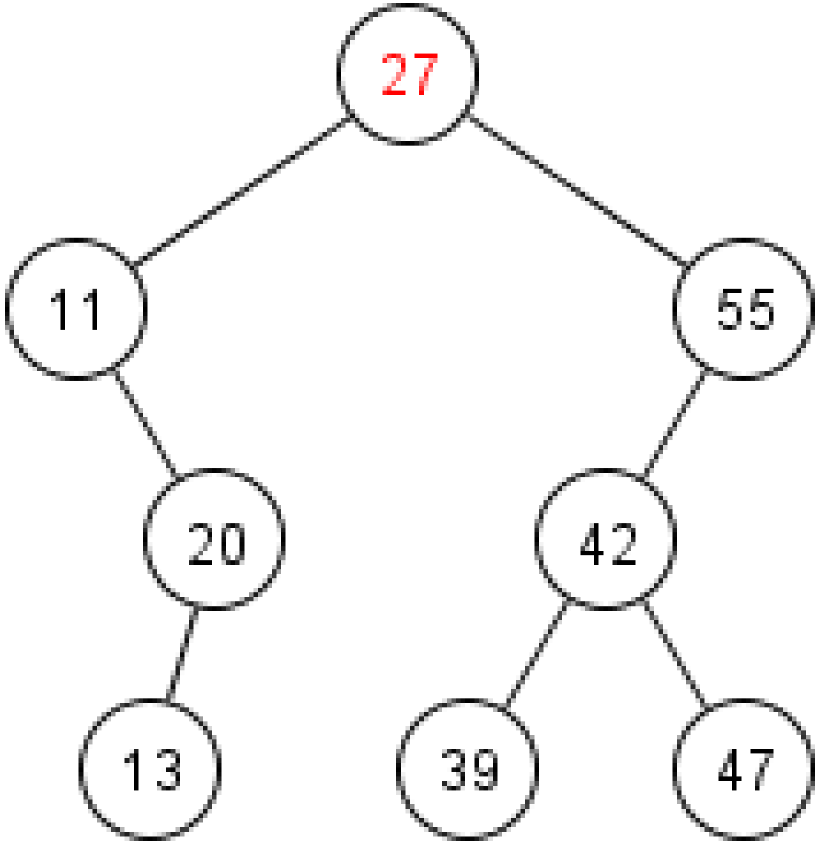
Binäre Suchbäume

23 löschen:



Binäre Suchbäume

23 löschen:



Binäre Suchbäume

Drei Fälle beim Löschen von *x*:
- x ist Blattknoten – einfach entfernen
- x hat genau ein Kind z – x entfernen und Vater von x mit z verbinden
- x hat zwei Kinder
    - Nachfolger z von x hat nur ein Kind
    - ersetze x durch z
    - lösche z

void delete(*Knoten* x, *Schlüssel* k)
if x=TNULL **then** *x not in tree*
else if x.key > z.key **then** delete(x.left, z)
else if x.key < z.key **then** delete(x.right, z)
else if x.left = TNULL **then** x = x.right
else if x.right = TNULL **then** x = x.left
else z = min(x.right)
    copy data from z to x
    delete(x.right, z.key)

Laufzeit search, insert, delete: O(h), h = Höhe des Baumes
Algorithmen durchläuft maximal einen Pfad von der Wurzel zu einem (leeren) Blatt

Wie können wir einen balancierten binärer Suchbaum konstruieren?

AVL - Bäume

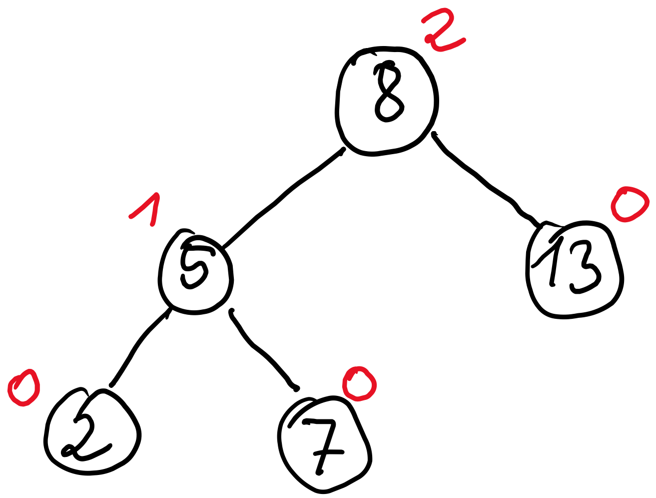
Name von den beiden Erfindern **A**delson-**V**elskii und **L**andis (1962)
erste Methode fürs Balancieren von Suchbäumen; inzwischen gibt es unzählige weitere

Definition:

Ein binärer Suchbaum *T* heißt AVL-Baum, wenn sich bei jedem Knoten *v* von *T* die Höhen der beiden Unterbäume um höchstens 1 unterscheiden.

Wir speichern Höhe *h(v)* für jeden Knoten *v*

h(v)



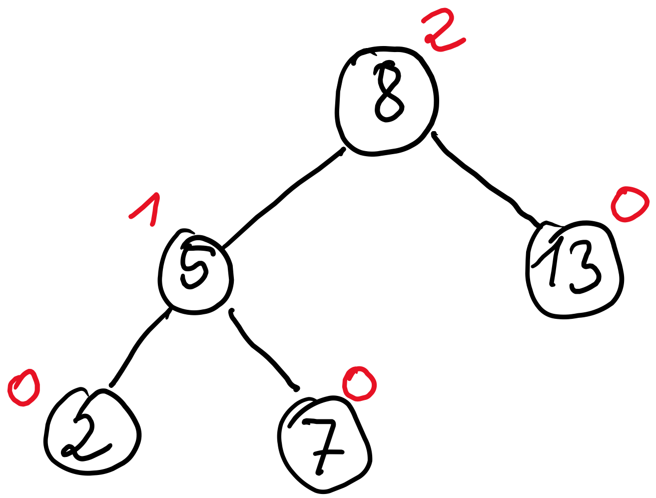
AVL - Bäume

Name von den beiden Erfindern **A**delson-**V**elskii und **L**andis (1962)
erste Methode fürs Balancieren von Suchbäumen; inzwischen gibt es unzählige weitere

> **Definition:**
> Ein binärer Suchbaum *T* heißt AVL-Baum, wenn sich bei jedem Knoten *v* von *T* die Höhen der beiden Unterbäume um höchstens 1 unterscheiden.

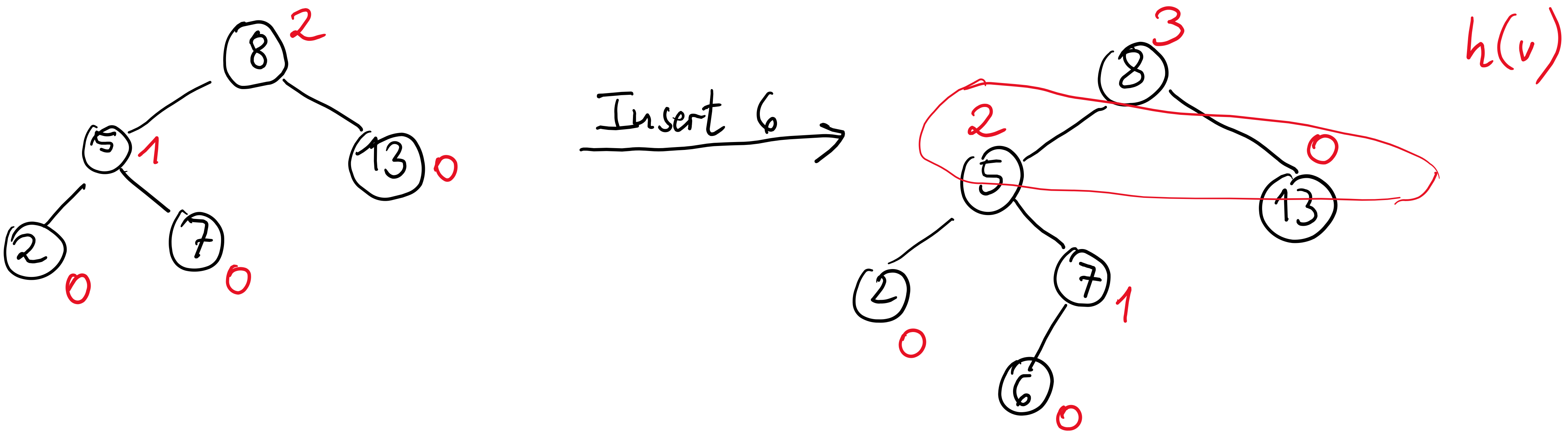
Wir speichern Höhe *h(v)* für jeden Knoten *v*

h(v)



AVL – Bäume: Operationen

- **Aufzählen**: kann direkt von binären Suchbäumen übernommen werden (traversieren)
- **Suche**: kann direkt von binären Suchbäumen übernommen werden
- **Einfügen**: binäre Suchbaummethode kann zu Problemen führen (AVL Eigenschaft wird evlt. verletzt)
- **Löschen**: binäre Suchbaummethode kann zu Problemen führen (AVL Eigenschaft wird evlt. verletzt)



AVL – Bäume: Operationen

Einfügen oder **Löschen** eines Elementes in einem binären Suchbaum kann Höhe eines Teilbaums höchstens um 1 verändern.

Der schlimmste auftretende Höhenunterschied ist also 2.

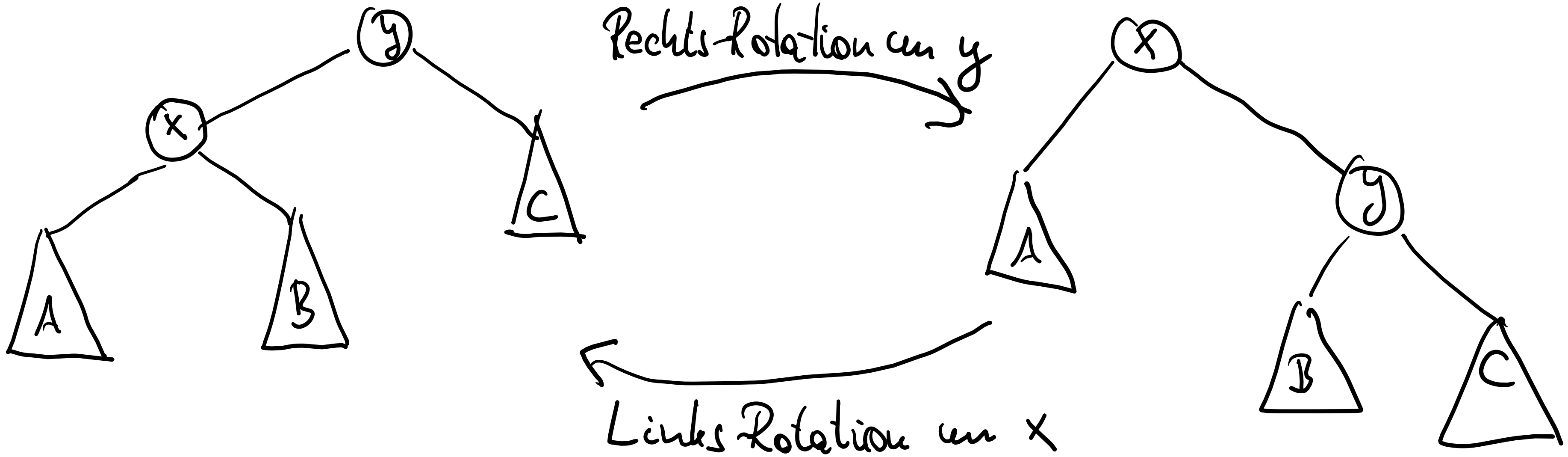
Definition: Ein binärer Suchbaum heißt *Beinahe-AVL-Baum*, wenn bei allen Knoten des Baums der Höhenunterschied der beiden Unterbäume höchstens 1 ist, außer bei der Wurzel, wo er 2 beträgt.

Ziel:

Schnelles Überführen eines Beinahe-AVL-Baums in einen echten AVL-Baum durch lokale Umstrukturierung („Rotation“)

Rotationen in Suchbäumen

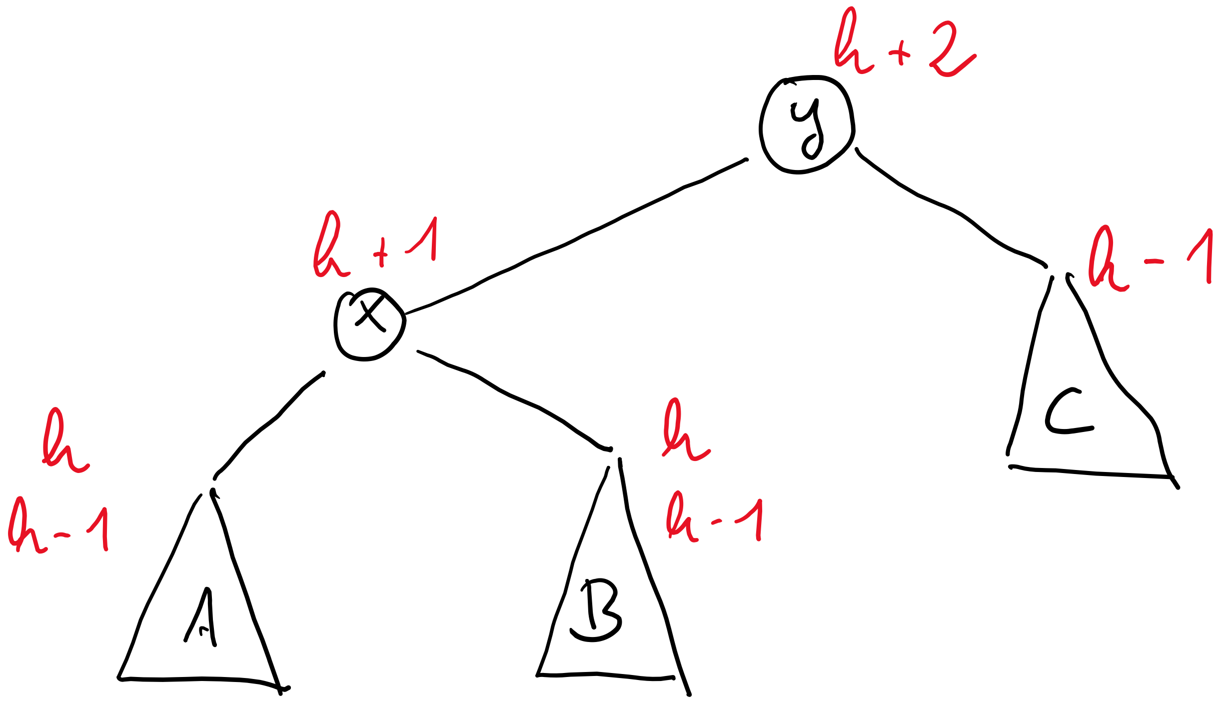
Eine Rotation in einem Suchbaum ist eine lokale Umstrukturierung des Baums, sodass duch Suchbaumeigenschart (Schlüsselordnung) erhalten bleibt.



Knoten x und y müssen beide existieren, damit die jeweilige Rotation angewandt werden kann.

Balancieren: Beinahe-AVL-Baum in AVL-Baum überführen

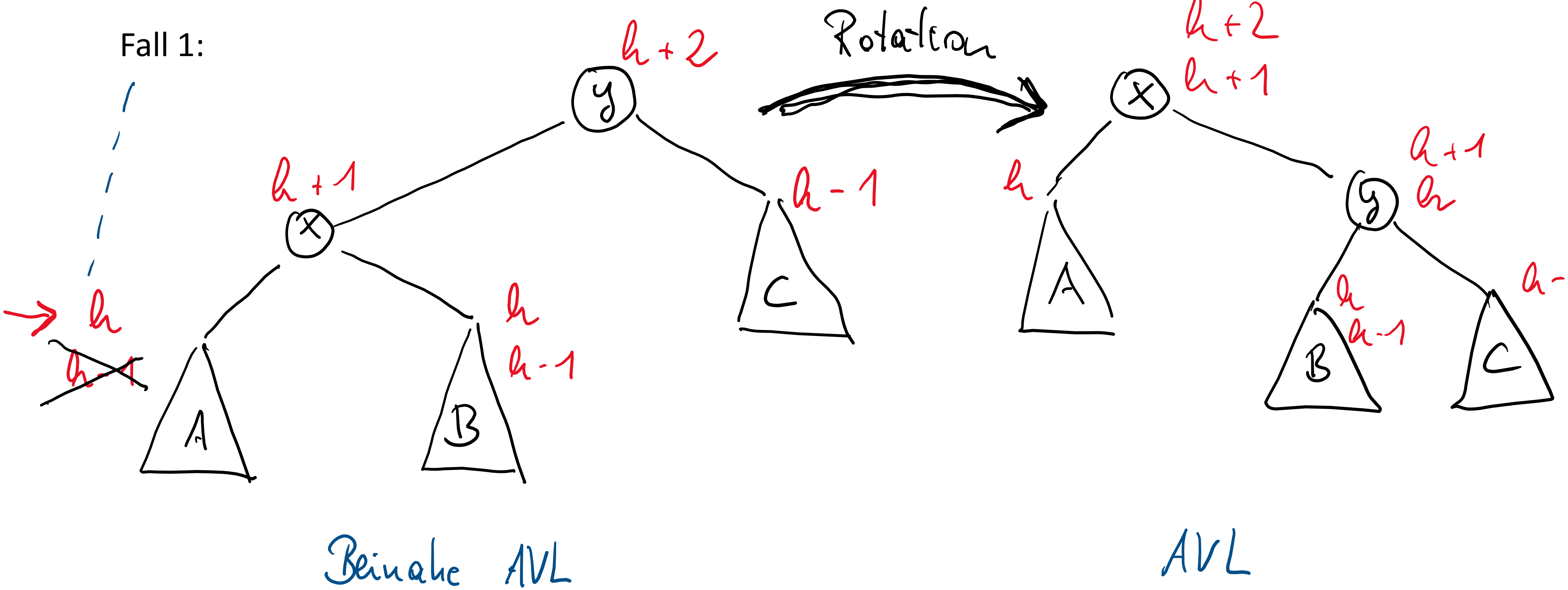
Annahme: linker Teilbaum höher als der rechte; der ander Fall ist symmetrisch



Balancieren: Beinahe-AVL-Baum in AVL-Baum überführen

Annahme: linker Teilbaum höher als der rechte; der ander Fall ist symmetrisch

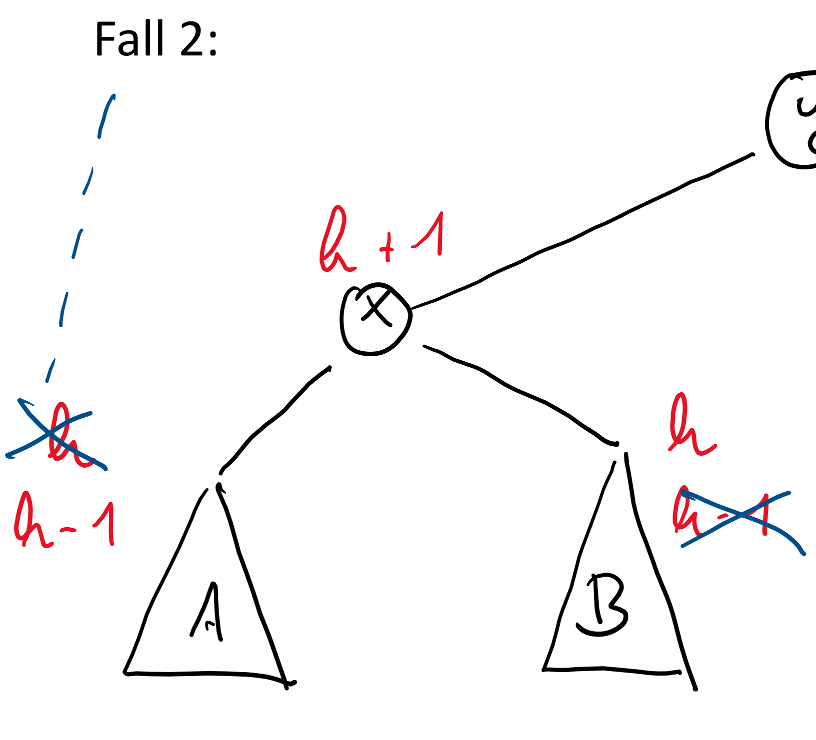
Fall 1:



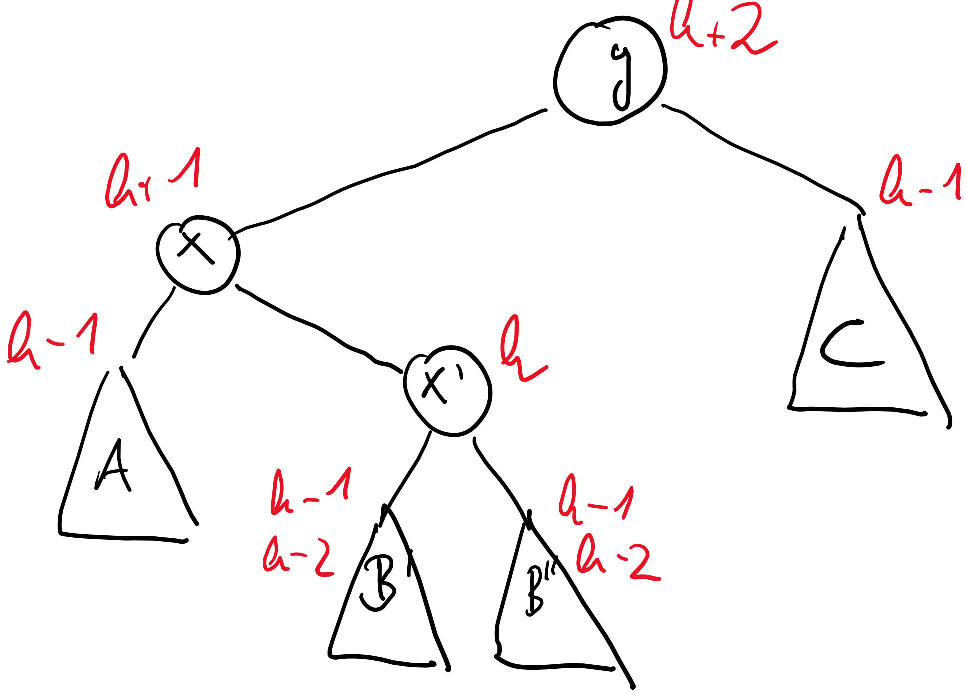
Balancieren: Beinahe-AVL-Baum in AVL-Baum überführen

Annahme: linker Teilbaum höher als der rechte; der ander Fall ist symmetrisch

Fall 2:



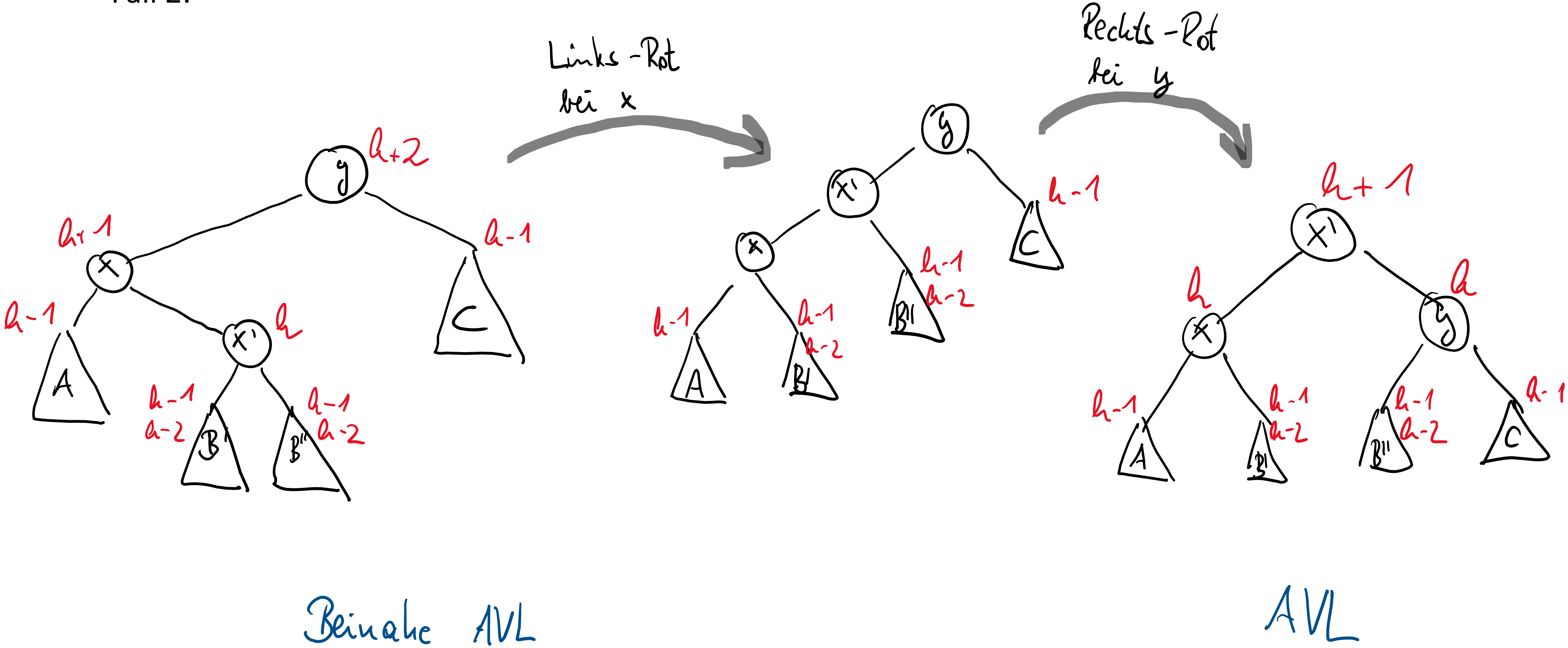
Beinahe AVL



mit mehr Detail

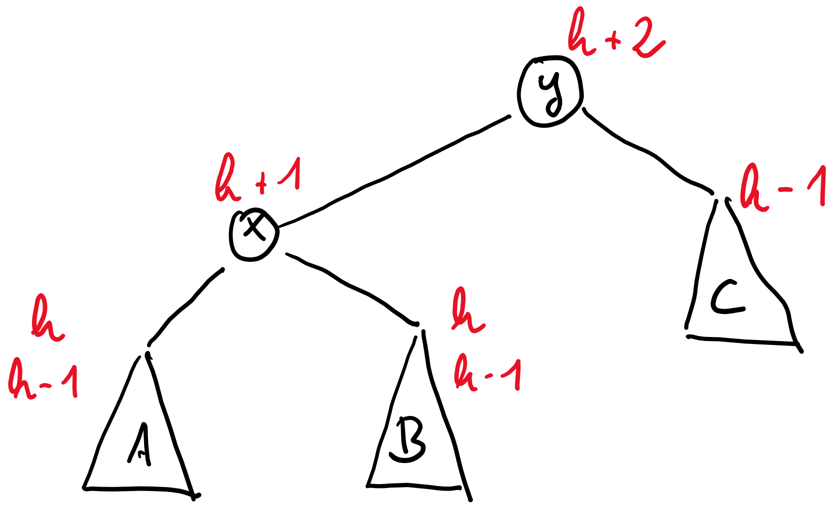
Balancieren: Beinahe-AVL-Baum in AVL-Baum überführen

Fall 2:



Balancieren: Beinahe-AVL-Baum in AVL-Baum überführen

Annahme: linker Teilbaum höher als der rechte; der ander Fall ist symmetrisch



Fall 1: *A* hat Höhe *h*
⇒ *B* hat Höhe *h* oder *h-1*
⇒ Rechts-Rotation um y ergibt AVL-Baum

Fall 2: *A* hat Höhe *h-1*
⇒ *B* hat Höhe *h*
⇒ 1. Links-Rotation um x
2. Rechts-Rotation um y ergibt AVL-Baum

Balancieren: Beinahe-AVL-Baum in AVL-Baum überführen

Lemma: Ein Beinahe-AVL-Baum der Höhe H kann in konstanter Zeit mit mittels höchstens zweier Rotationen zu eine AVL-Baum der Höhe H oder H-1 balanciert werden.

Die entsprechende Prozedur sei void Balance(Knoten x)

Einfügen in AVL-Baum

Gleich wie in normalem binären Suchbaum, aber wende Balance(x) auf jeden Knoten auf dem Einfügepfad an.   (mit Effekt bei höchstens einem Knoten)

void AVL-insert (*Knoten* x, *Schlüssel k*)
if x=TNULL **then** x = makenode(*k*)
else if k < x.key **then** AVL-insert(x.left, *k*)
else if k > x.key **then** AVL-insert(x.right, *k*)
else *key k already in tree*
Balance(x)

Löschen in AVL-Baum

Gleich wie in normalem binären Suchbaum, aber wende Balance(x) auf jeden Knoten auf dem Löschpfad an.   (mit Effekt bei möglicherweise vielen Knoten auf dem Pfad)

void AVL-delete(*Knoten* x, *Schlüssel* k)
if x=TNULL **then** *x not in tree*
else if x.key > z.key **then** AVL-delete(x.left, z)
else if x.key < z.key **then** AVL-delete(x.right, z)
else if x.left = TNULL **then** x = x.right
else if x.right = TNULL **then** x = x.left
else z = min(x.right)
        copy data from z to x
        AVL-delete(x.right, z.key)
Balance(x)

AVL-Bäume

Satz: AVL-Bäume haben logarithmische Tiefe. Sie erlauben Suche, Einfügen und Löschen in logarithmischer Zeit und verwenden nur linearen Speicherplatz. Alle im Baum gespeicherten Elemente können in linearer Zeit aufgezählt werden.