12. Balancierte binäre Suchbäume

Binäre Suchbäume:

- Ausgabe aller Elemente in O(n)
- Suche, Minimum, Maximum, Nachfolger in O(h)
- Einfügen, Löschen in O(h)

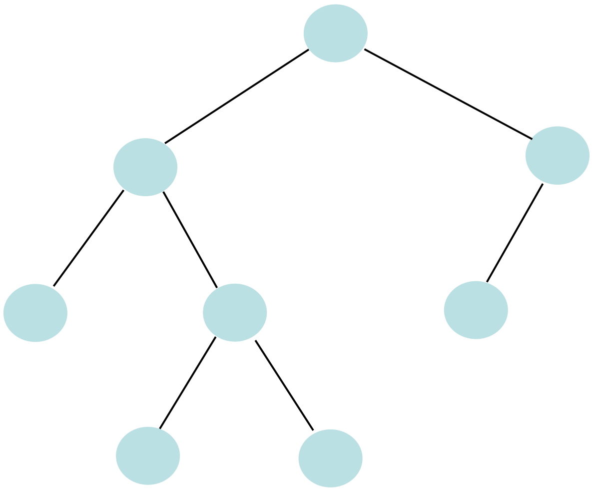
Frage:

- Wie kann man eine „kleine“ Höhe unter Einfügen und Löschen garantieren?

Balancierte binäre Suchbäume

AVL-Bäume

- Ein Baum heißt AVL-Baum, wenn für jeden Knoten gilt: Die Höhe seines linken und rechten Teilbaums unterscheidet sich höchstens um 1.



Balancierte binäre Suchbäume

Satz 12.1

Für jeden AVL-Baum der Höhe h mit n Knoten gilt:

$$(3/2)^h \leq n \leq 2^{h+1}-1$$

Balancierte binäre Suchbäume

Satz 12.1

Für jeden AVL-Baum der Höhe h mit n Knoten gilt:

$$(3/2)^h \leq n \leq 2^{h+1}-1$$

Beweis: Tafel

Korollar 12.2

Ein AVL-Baum mit n Knoten hat Höhe $\Theta(\log n)$.

Balancierte binäre Suchbäume

Dynamische AVL-Bäume

- Operationen Suche, Einfügen, Löschen, Min/Max, Vorgänger/Nachfolger,… wie für binäre Suchbäume
- Laufzeit O(h) für diese Operationen
- Nur Einfügen/Löschen verändern Struktur des Baums

Problem:

- Wir brauchen Prozedur, um AVL-Eigenschaft nach Einfügen/Löschen wiederherzustellen.

Balancierte binäre Suchbäume

Dynamische

Nach Korollar 12.2 gilt
$h = \Theta(\log n)$

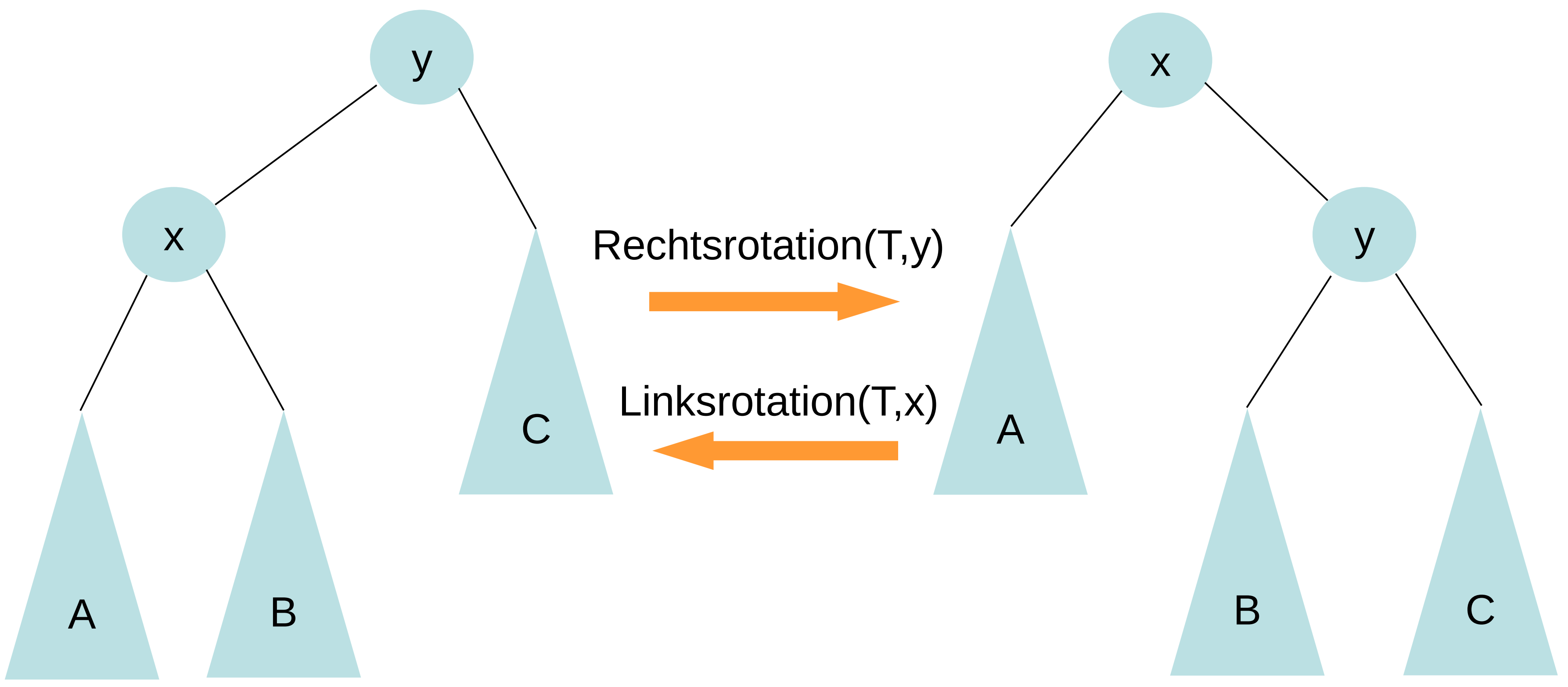
- Operationen S… , Min/Max, Vorgänger/Na… wie für binäre Suchbäume
- Laufzeit O(h) für diese Operationen
- Nur Einfügen/Löschen verändern Struktur des Baums

Problem:

- Wir brauchen Prozedur, um AVL-Eigenschaft nach Einfügen/Löschen wiederherzustellen.

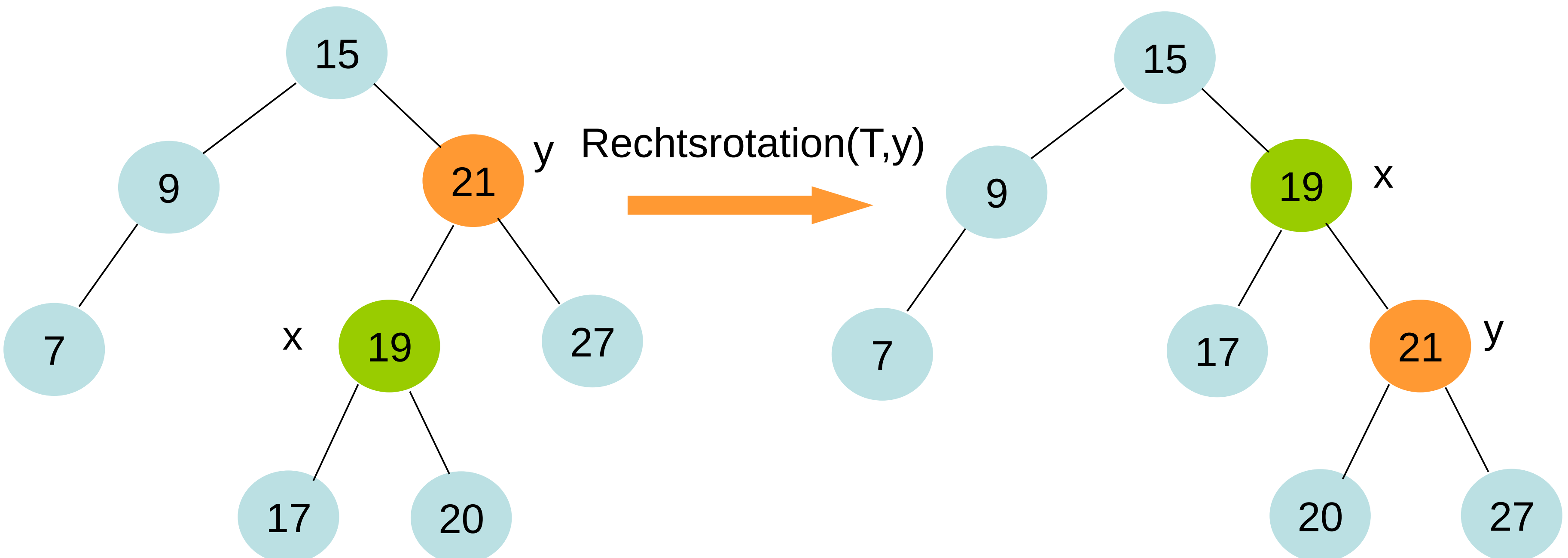
Balancierte binäre Suchbäume

Rotationen, die Suchbaumeigenschaft bewahren:

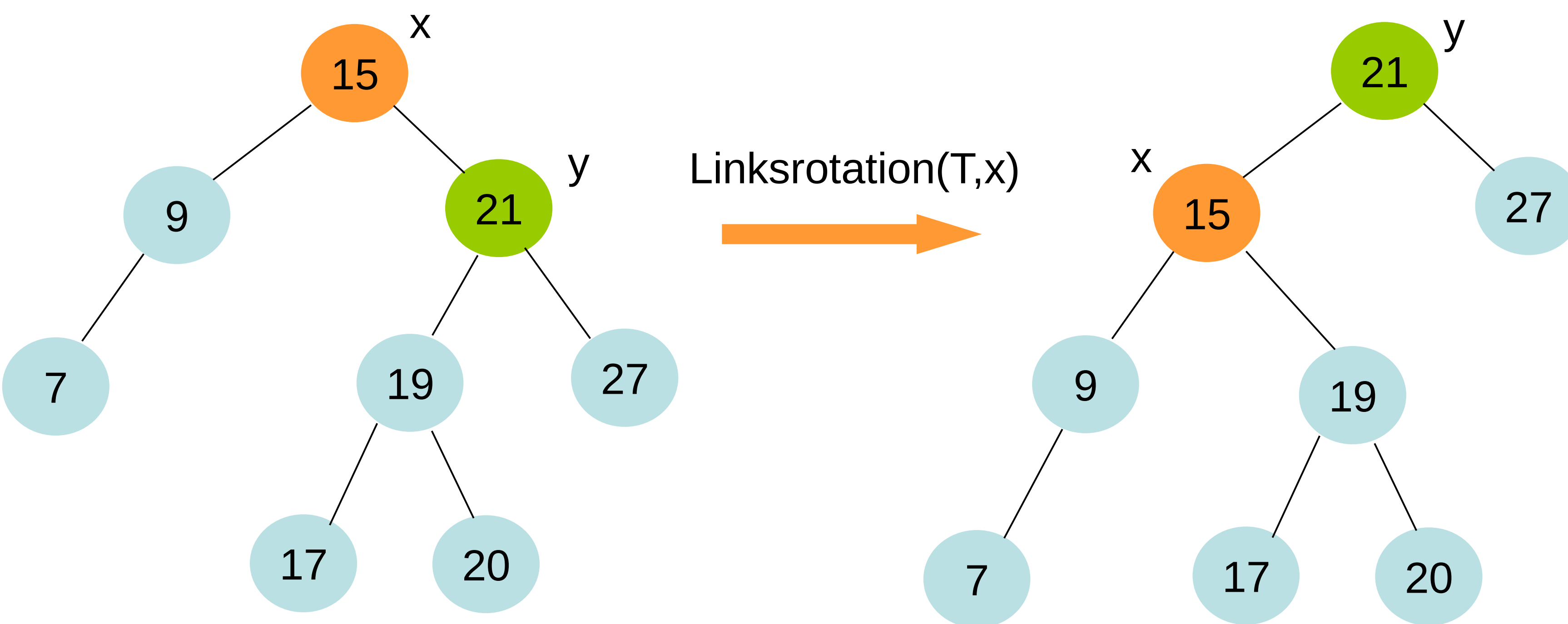


Balancierte binäre Suchbäume

Beispiel 1:



Beispiel 2:



Balancierte binäre Suchbäume

Linksrotation(T,x)

1. y $\leftarrow$ rc[x]
2. rc[x] $\leftarrow$ lc[y]
3. **if** lc[y]$\neq$**nil then** p[lc[y]] $\leftarrow$ x
4. p[y] $\leftarrow$ p[x]
5. **if** p[x]=**nil then** root[T] $\leftarrow$ y
6. **else if** x=lc[p[x]] **then** lc[p[x]] $\leftarrow$ y
7.   **else** rc[p[x]] $\leftarrow$ y
8. lc[y] $\leftarrow$ x
9. p[x] $\leftarrow$ y
10. h[x] $\leftarrow$ 1+ max{h[lc[x]],h[rc[x]]}
11. h[y] $\leftarrow$ 1+ max{h[lc[y]],h[rc[y]]}

Balancierte binäre Suchbäume

Linksrotation(T,x)

1. y $\leftarrow$ rc[x]
2. rc[x] $\leftarrow$ lc[y]
3. **if** lc[y]$\neq$**nil then** p[lc[y]] $\leftarrow$ x
4. p[y] $\leftarrow$ p[x]
5. **if** p[x]=**nil then** root[T] $\leftarrow$ y
6. **else if** x=lc[p[x]] **then** lc[p[x]] $\leftarrow$ y
7.   **else** rc[p[x]] $\leftarrow$ y
8. lc[y] $\leftarrow$ x
9. p[x] $\leftarrow$ y
10. h[x] $\leftarrow$ 1+ max{h[lc[x]],h[rc[x]]}
11. h[y] $\leftarrow$ 1+ max{h[lc[y]],h[rc[y]]}

Im folgenden Beispiel werden die Zeilen 10 und 11 nicht betrachtet

Balancierte binäre Suchbäume

Annahme: x hat rechtes Kind.

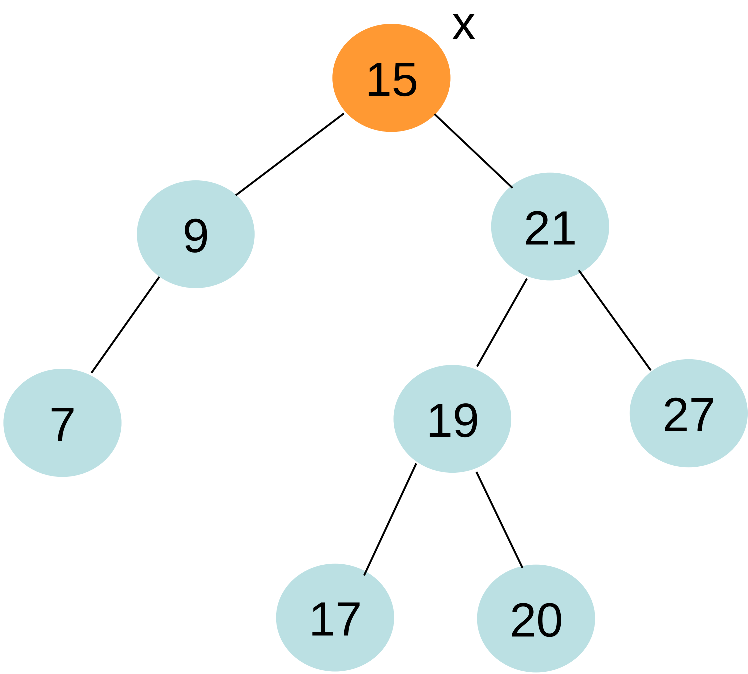
Linksrotation(T,x)

1. y ← rc[x]
2. rc[x] ← lc[y]
3. **if** lc[y]≠**nil then** p[lc[y]] ← x
4. p[y] ← p[x]
5. **if** p[x]=**nil then** root[T] ← y
6. **else if** x=lc[p[x]] **then** lc[p[x]] ← y
7.   **else** rc[p[x]] ← y
8. lc[y] ← x
9. p[x] ← y

Balancierte binäre Suchbäume

Linksrotation(T,x)

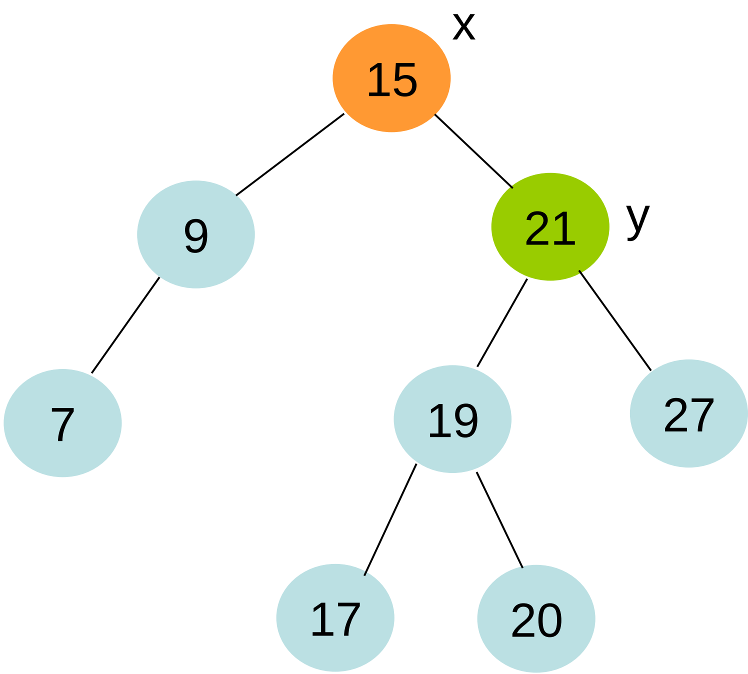
1. y $\leftarrow$ rc[x]
2. rc[x] $\leftarrow$ lc[y]
3. **if** lc[y]$\neq$**nil then** p[lc[y]] $\leftarrow$ x
4. p[y] $\leftarrow$ p[x]
5. **if** p[x]=**nil then** root[T] $\leftarrow$ y
6. **else if** x=lc[p[x]] **then** lc[p[x]] $\leftarrow$ y
7.   **else** rc[p[x]] $\leftarrow$ y
8. lc[y] $\leftarrow$ x
9. p[x] $\leftarrow$ y



Balancierte binäre Suchbäume

Linksrotation(T,x)

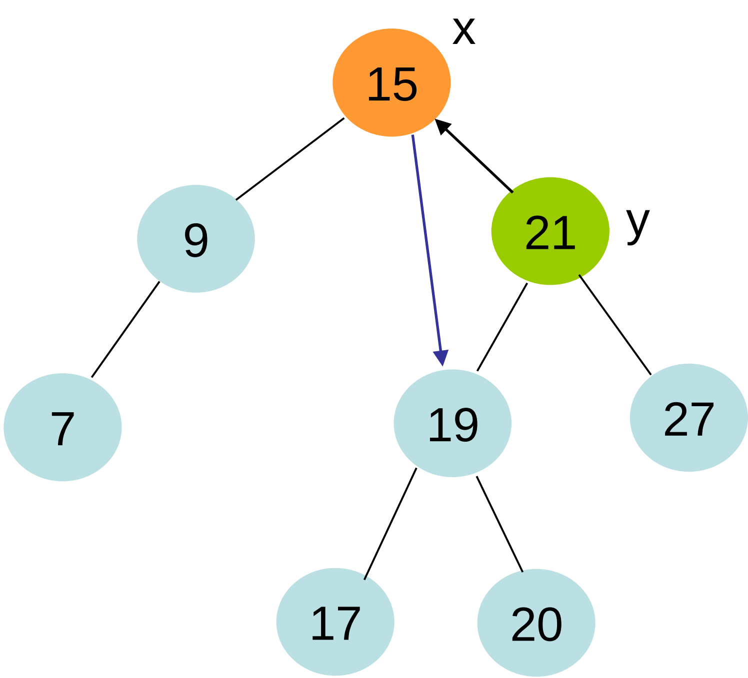
1. y $\leftarrow$ rc[x]
2. rc[x] $\leftarrow$ lc[y]
3. **if** lc[y]$\neq$**nil then** p[lc[y]] $\leftarrow$ x
4. p[y] $\leftarrow$ p[x]
5. **if** p[x]=**nil then** root[T] $\leftarrow$ y
6. **else if** x=lc[p[x]] **then** lc[p[x]] $\leftarrow$ y
7.   **else** rc[p[x]] $\leftarrow$ y
8. lc[y] $\leftarrow$ x
9. p[x] $\leftarrow$ y



Balancierte binäre Suchbäume

Linksrotation(T,x)

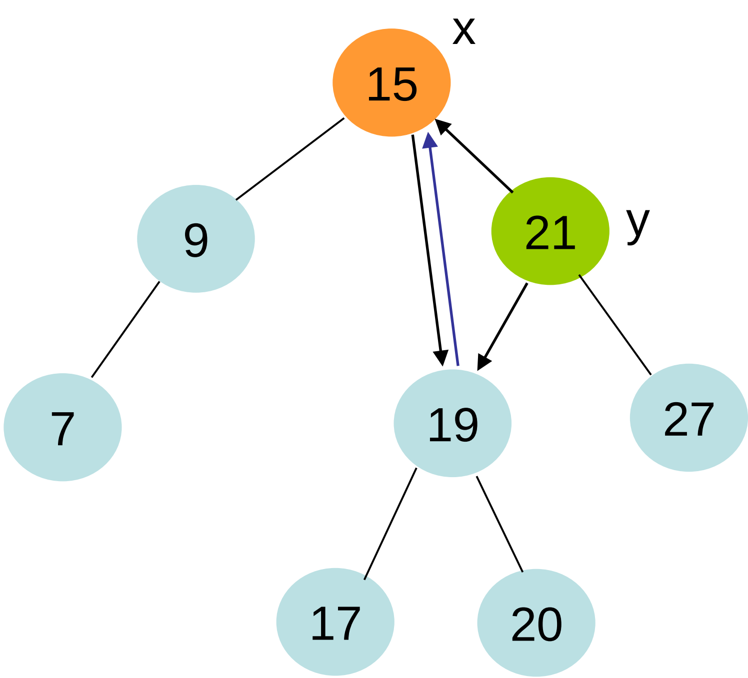
1. y $\leftarrow$ rc[x]
2. rc[x] $\leftarrow$ lc[y]
3. **if** lc[y]$\neq$**nil then** p[lc[y]] $\leftarrow$ x
4. p[y] $\leftarrow$ p[x]
5. **if** p[x]=**nil then** root[T] $\leftarrow$ y
6. **else if** x=lc[p[x]] **then** lc[p[x]] $\leftarrow$ y
7. **else** rc[p[x]] $\leftarrow$ y
8. lc[y] $\leftarrow$ x
9. p[x] $\leftarrow$ y



Balancierte binäre Suchbäume

Linksrotation(T,x)

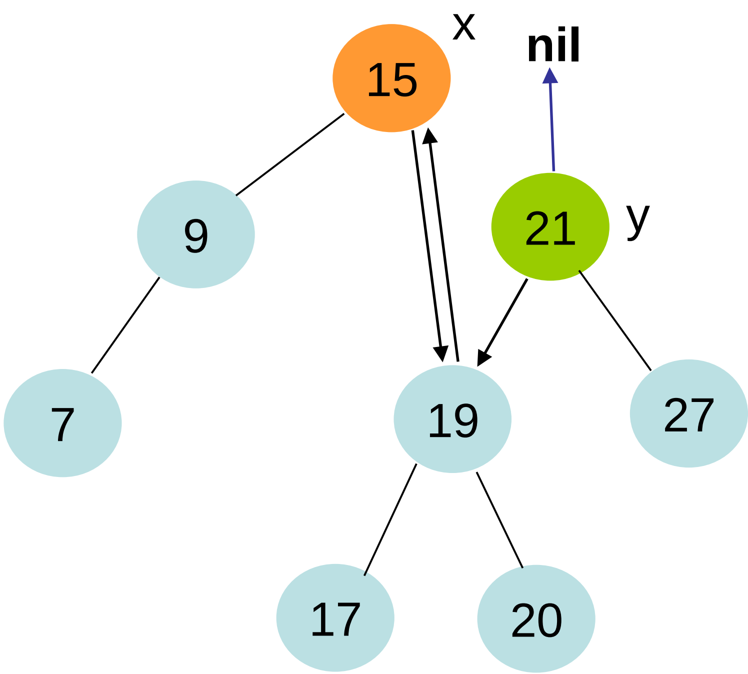
1. y ← rc[x]
2. rc[x] ← lc[y]
3. **if** lc[y]≠**nil then** p[lc[y]] ← x
4. p[y] ← p[x]
5. **if** p[x]=**nil then** root[T] ← y
6. **else if** x=lc[p[x]] **then** lc[p[x]] ← y
7. **else** rc[p[x]] ← y
8. lc[y] ← x
9. p[x] ← y



Balancierte binäre Suchbäume

Linksrotation(T,x)

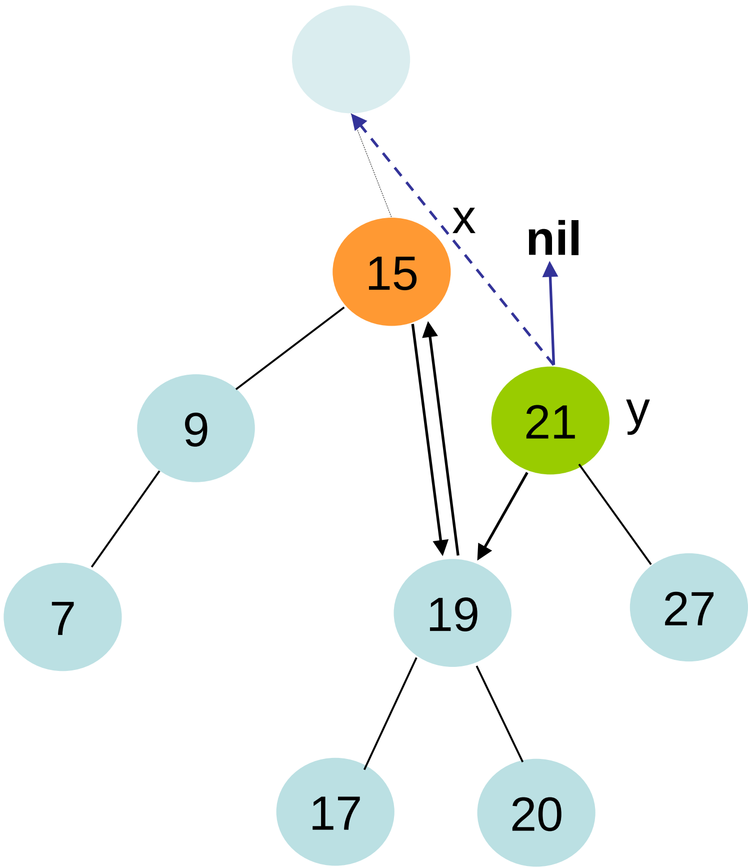
1. y ← rc[x]
2. rc[x] ← lc[y]
3. **if** lc[y]≠**nil then** p[lc[y]] ← x
4. p[y] ← p[x]
5. **if** p[x]=**nil then** root[T] ← y
6. **else if** x=lc[p[x]] **then** lc[p[x]] ← y
7.   **else** rc[p[x]] ← y
8. lc[y] ← x
9. p[x] ← y



Balancierte binäre Suchbäume

Linksrotation(T,x)

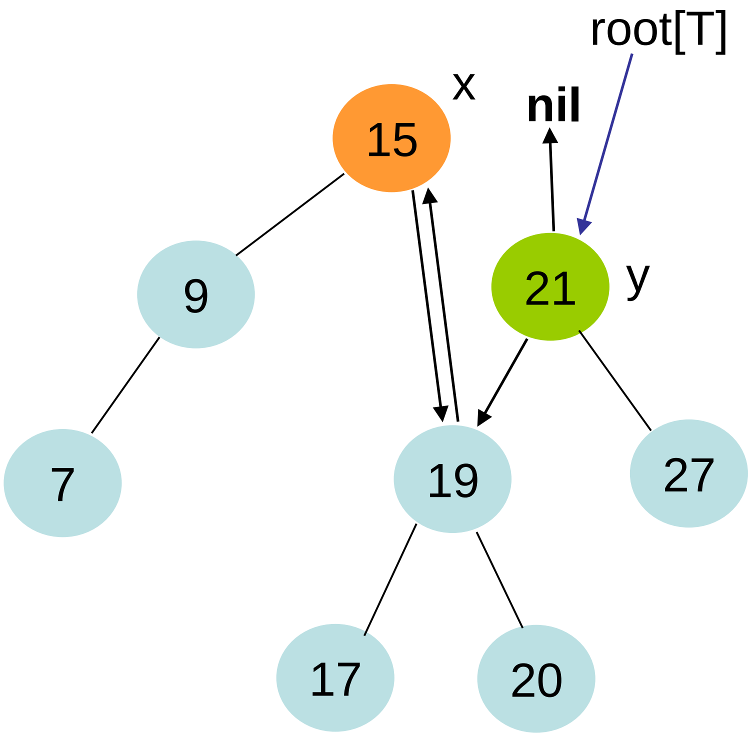
1. y ← rc[x]
2. rc[x] ← lc[y]
3. **if** lc[y]≠**nil then** p[lc[y]] ← x
4. p[y] ← p[x]
5. **if** p[x]=**nil then** root[T] ← y
6. **else if** x=lc[p[x]] **then** lc[p[x]] ← y
7. **else** rc[p[x]] ← y
8. lc[y] ← x
9. p[x] ← y



Balancierte binäre Suchbäume

Linksrotation(T,x)

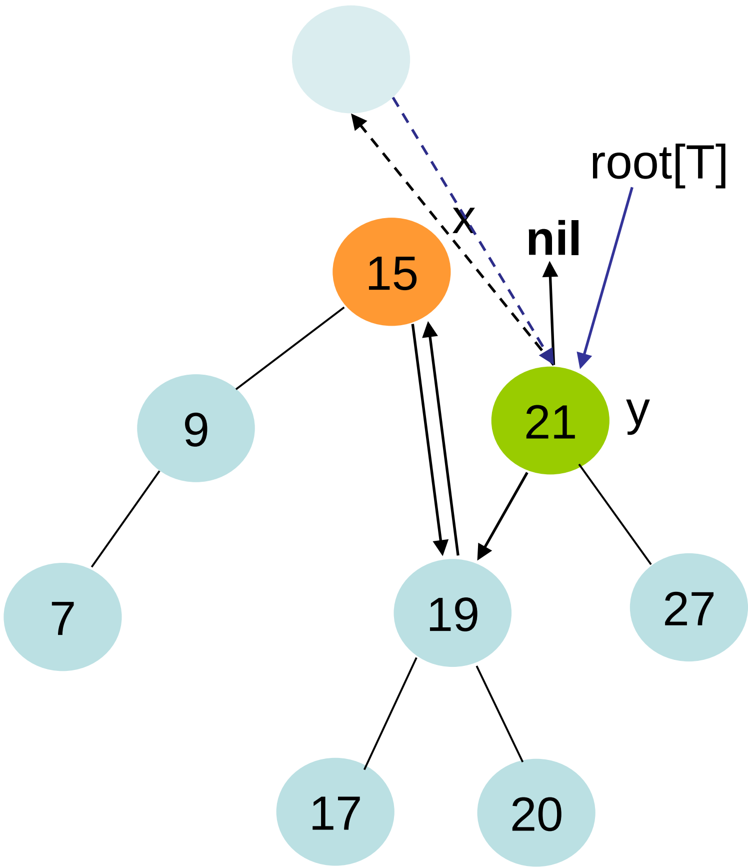
1. y $\leftarrow$ rc[x]
2. rc[x] $\leftarrow$ lc[y]
3. **if** lc[y]$\neq$**nil then** p[lc[y]] $\leftarrow$ x
4. p[y] $\leftarrow$ p[x]
5. **if** p[x]=**nil then** root[T] $\leftarrow$ y
6. **else if** x=lc[p[x]] **then** lc[p[x]] $\leftarrow$ y
7. **else** rc[p[x]] $\leftarrow$ y
8. lc[y] $\leftarrow$ x
9. p[x] $\leftarrow$ y



Balancierte binäre Suchbäume

Linksrotation(T,x)

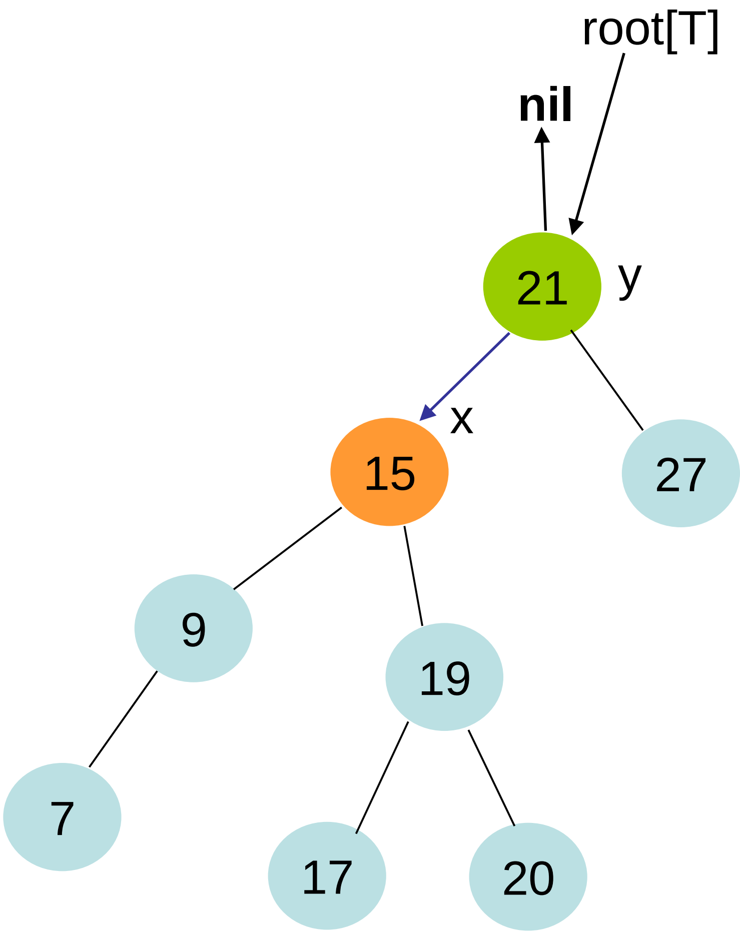
1. y ← rc[x]
2. rc[x] ← lc[y]
3. **if** lc[y]≠**nil** **then** p[lc[y]] ← x
4. p[y] ← p[x]
5. **if** p[x]=**nil** **then** root[T] ← y
6. **else if** x=lc[p[x]] **then** lc[p[x]] ← y
7. **else** rc[p[x]] ← y
8. lc[y] ← x
9. p[x] ← y



Balancierte binäre Suchbäume

Linksrotation(T,x)

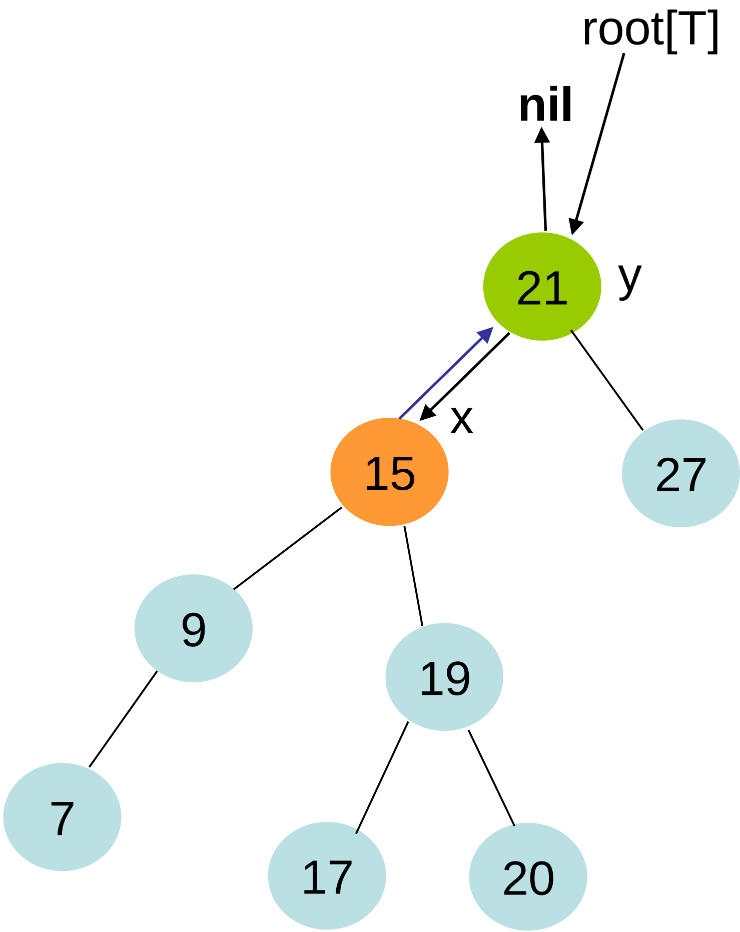
1. y ← rc[x]
2. rc[x] ← lc[y]
3. **if** lc[y]≠**nil then** p[lc[y]] ← x
4. p[y] ← p[x]
5. **if** p[x]=**nil then** root[T] ← y
6. **else if** x=lc[p[x]] **then** lc[p[x]] ← y
7. **else** rc[p[x]] ← y
8. lc[y] ← x
9. p[x] ← y



Balancierte binäre Suchbäume

Linksrotation(T,x)

1. y ← rc[x]
2. rc[x] ← lc[y]
3. **if** lc[y]≠**nil then** p[lc[y]] ← x
4. p[y] ← p[x]
5. **if** p[x]=**nil then** root[T] ← y
6. **else if** x=lc[p[x]] **then** lc[p[x]] ← y
7. **else** rc[p[x]] ← y
8. lc[y] ← x
9. p[x] ← y

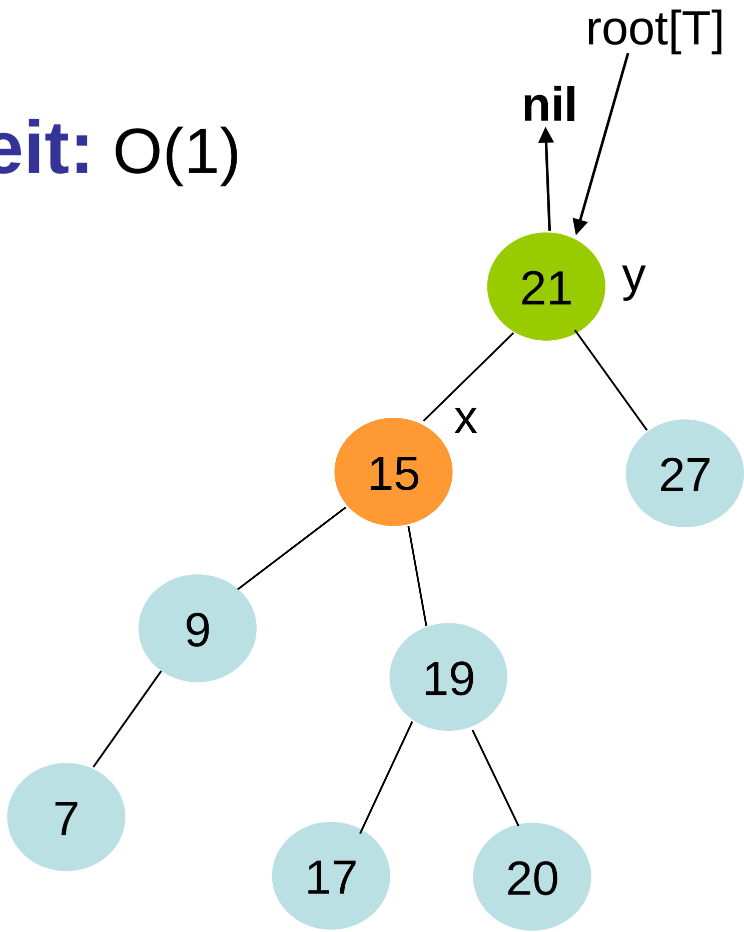


Balancierte binäre Suchbäume

Linksrotation(T,x)

1. y ← rc[x]
2. rc[x] ← lc[y]
3. **if** lc[y]≠**nil then** p[lc[y]] ← x
4. p[y] ← p[x]
5. **if** p[x]=**nil then** root[T] ← y
6. **else if** x=lc[p[x]] **then** lc[p[x]] ← y
7. **else** rc[p[x]] ← y
8. lc[y] ← x
9. p[x] ← y

Laufzeit: O(1)



Balancierte binäre Suchbäume

Zur Aufrechterhaltung der AVL-Baumeigenschaft betrachten wir zunächst ein vereinfachtes Problem.

Definition:

- Ein Baum heißt beinahe-AVL-Baum, wenn die AVL-Eigenschaft in jedem Knoten außer der Wurzel erfüllt ist und die Höhen der Unterbäume der Wurzel sich um höchstens 2 unterscheiden.

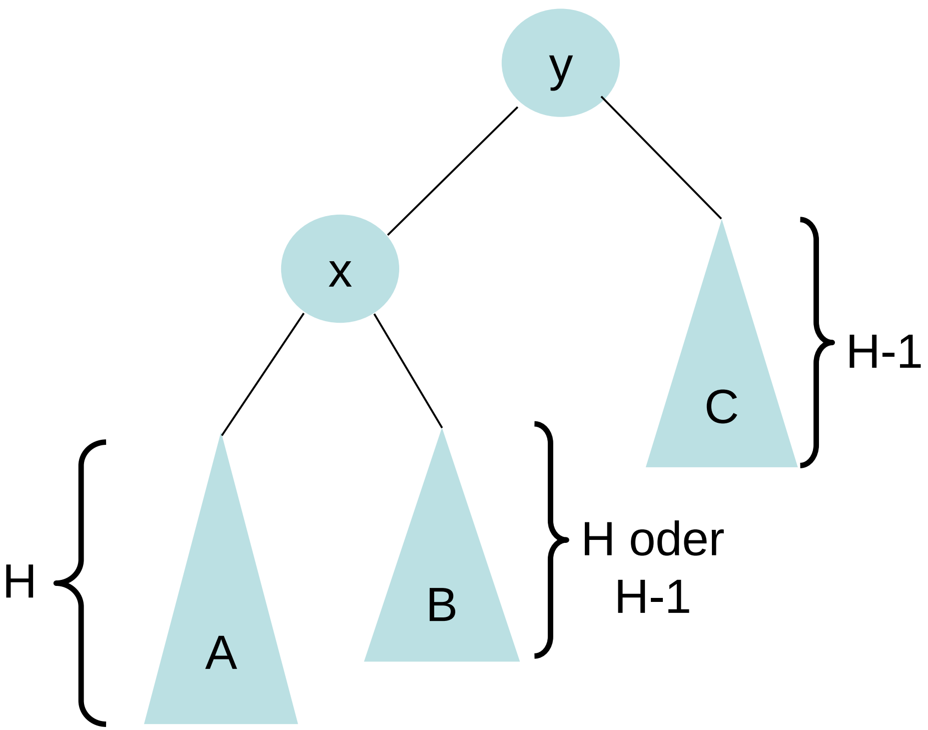
Balancierte binäre Suchbäume

Problem:

- Umformen eines beinahe-AVL-Baums in einen AVL-Baum mit Hilfe von Rotationen
- O.b.d.A.: Linker Teilbaum der Wurzel höher als der rechte (Fall für den rechten Teilbaum analog)

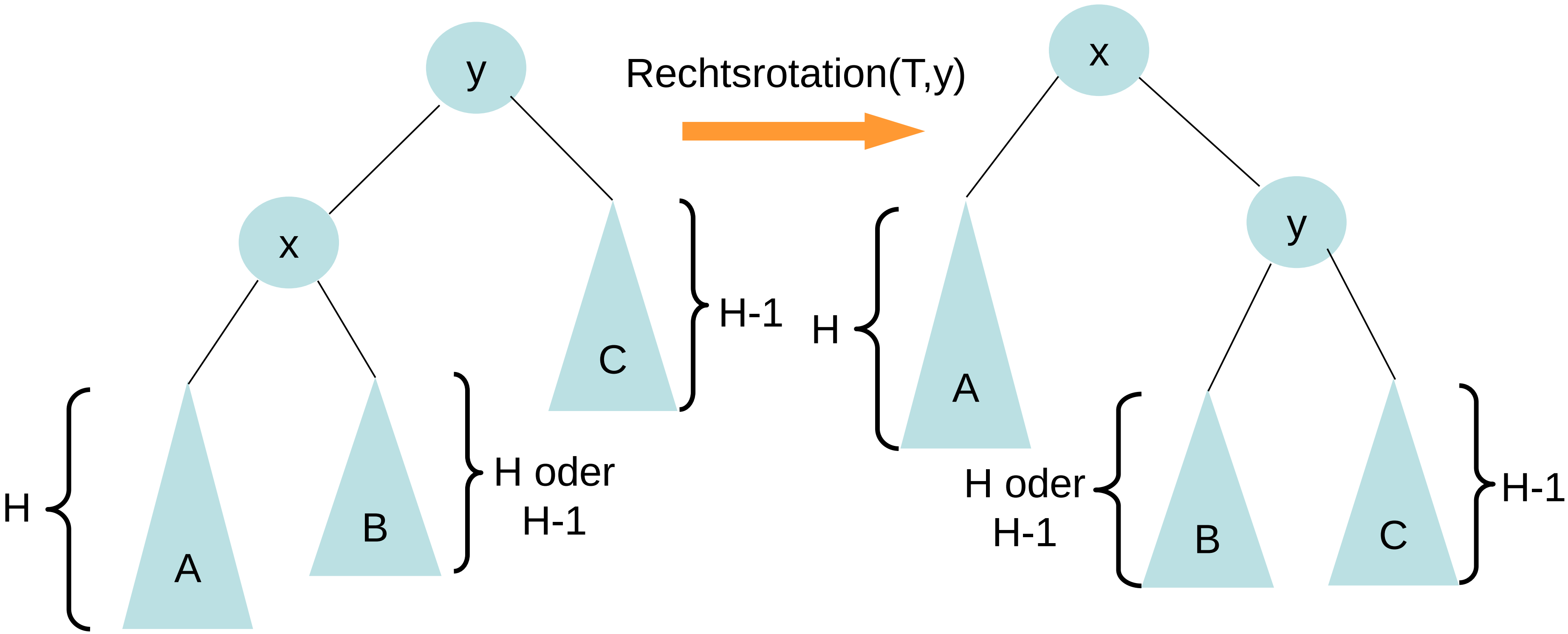
Balancierte binäre Suchbäume

Fall 1:



Balancierte binäre Suchbäume

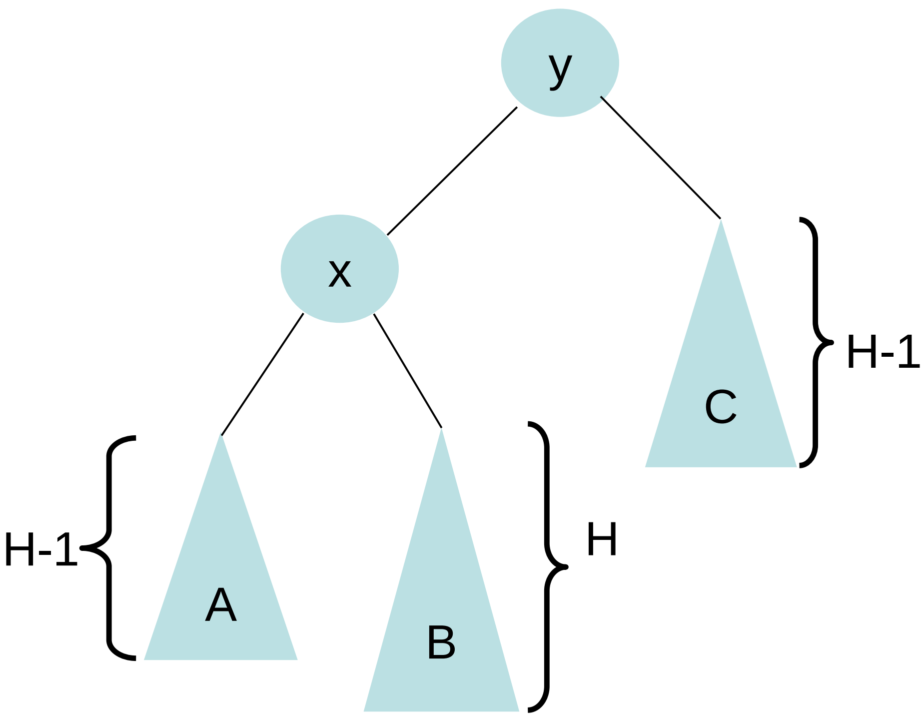
Fall 1:



Korrekter AVL-Baum

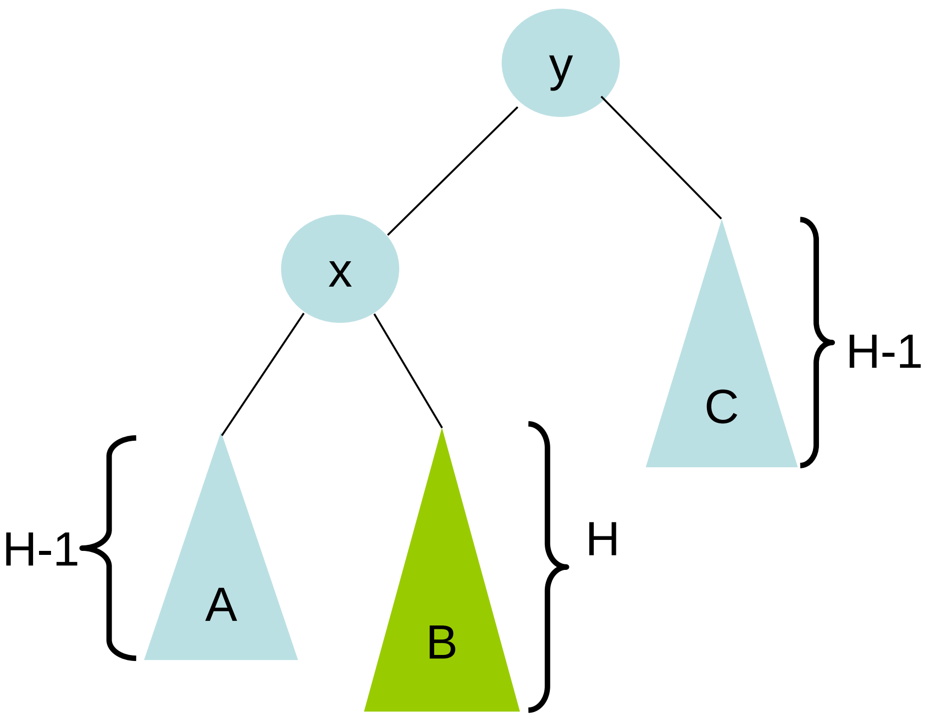
Balancierte binäre Suchbäume

Fall 2:



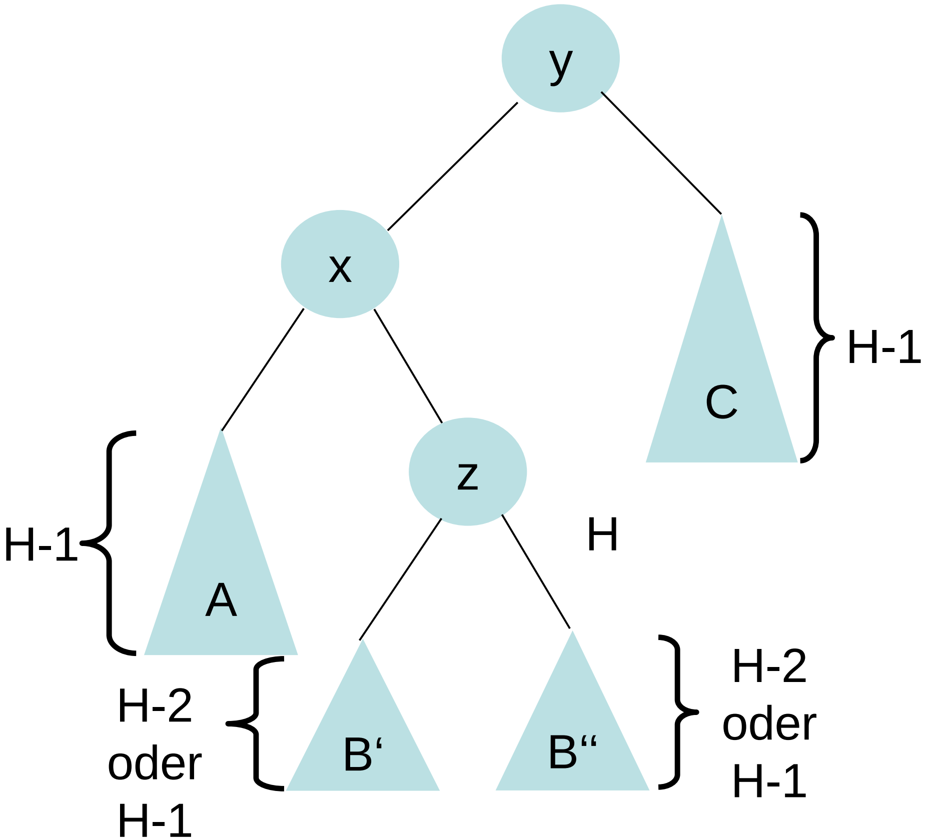
Balancierte binäre Suchbäume

Fall 2:



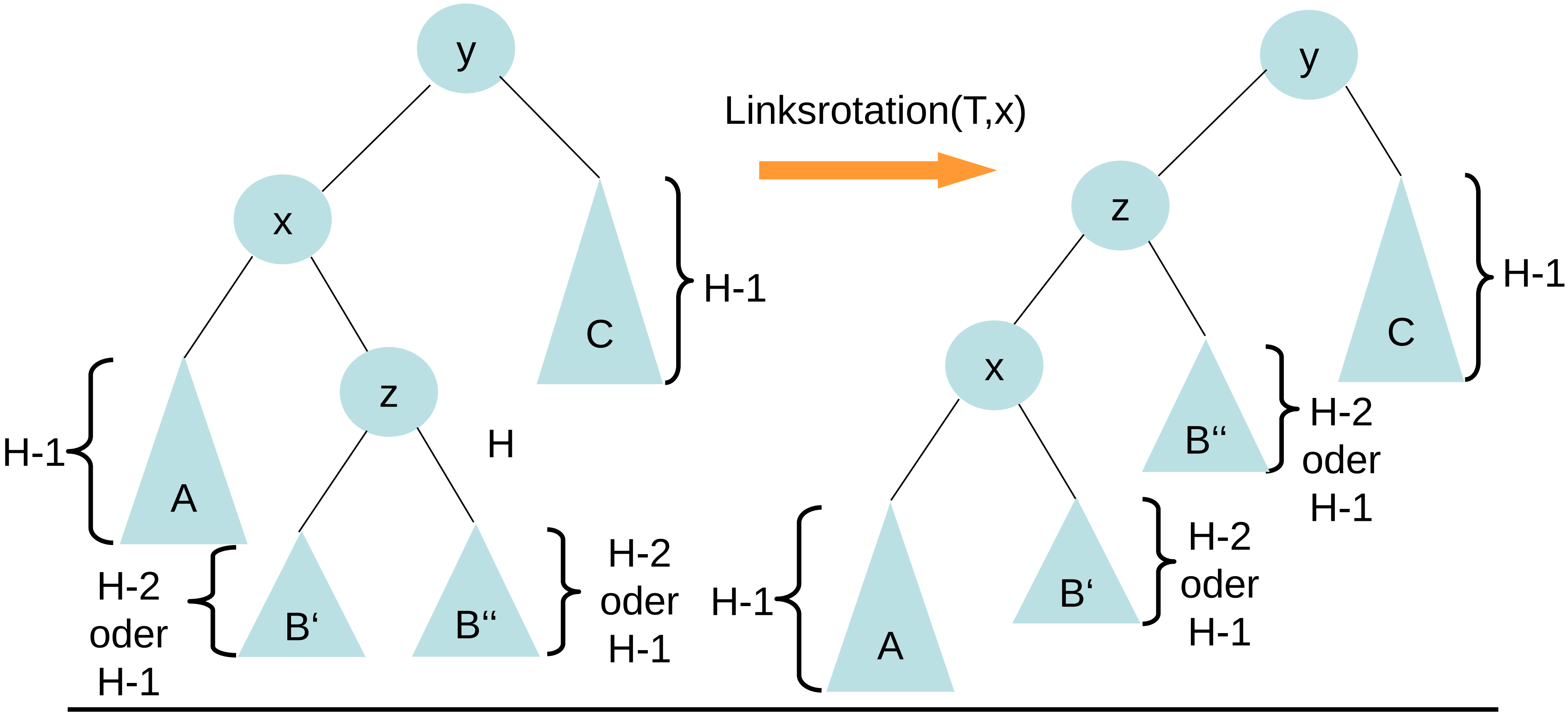
Balancierte binäre Suchbäume

Fall 2:



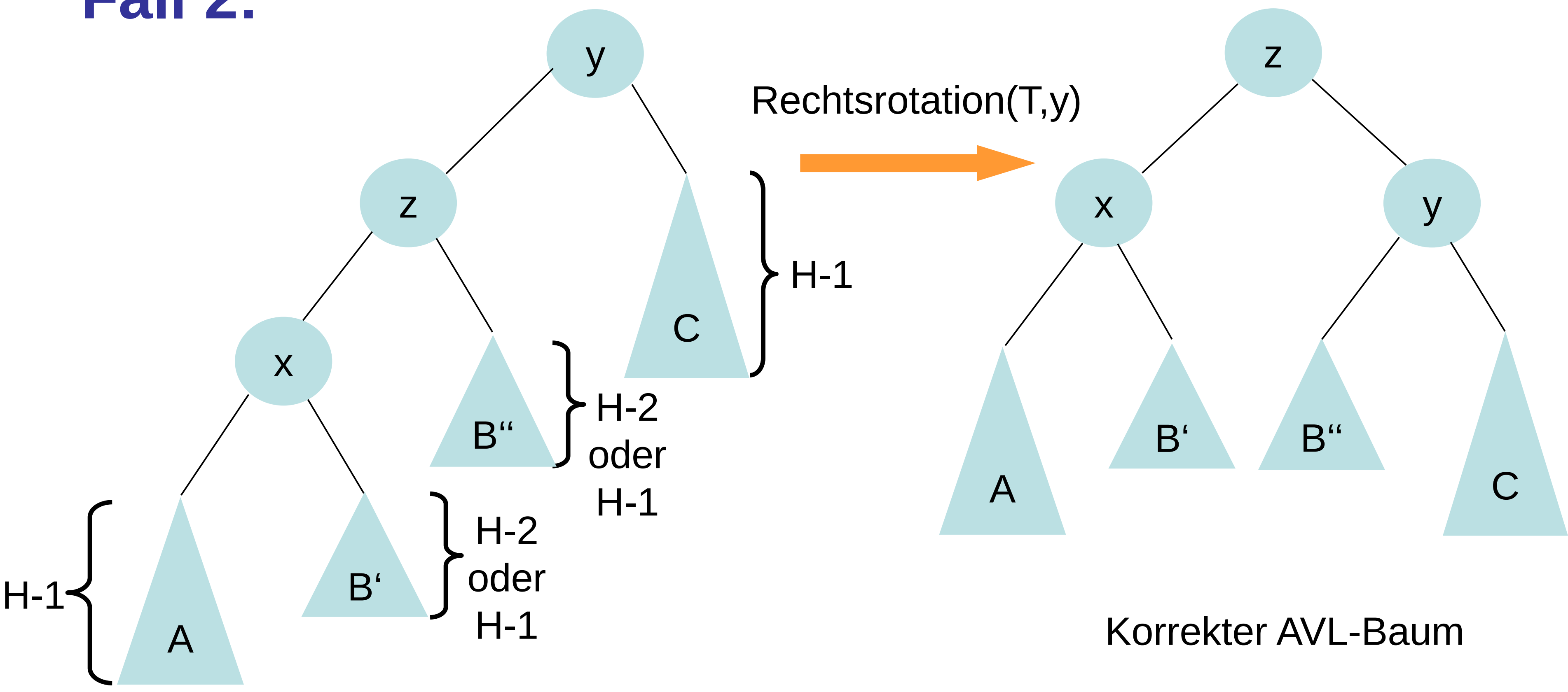
Balancierte binäre Suchbäume

Fall 2:



Balancierte binäre Suchbäume

Fall 2:

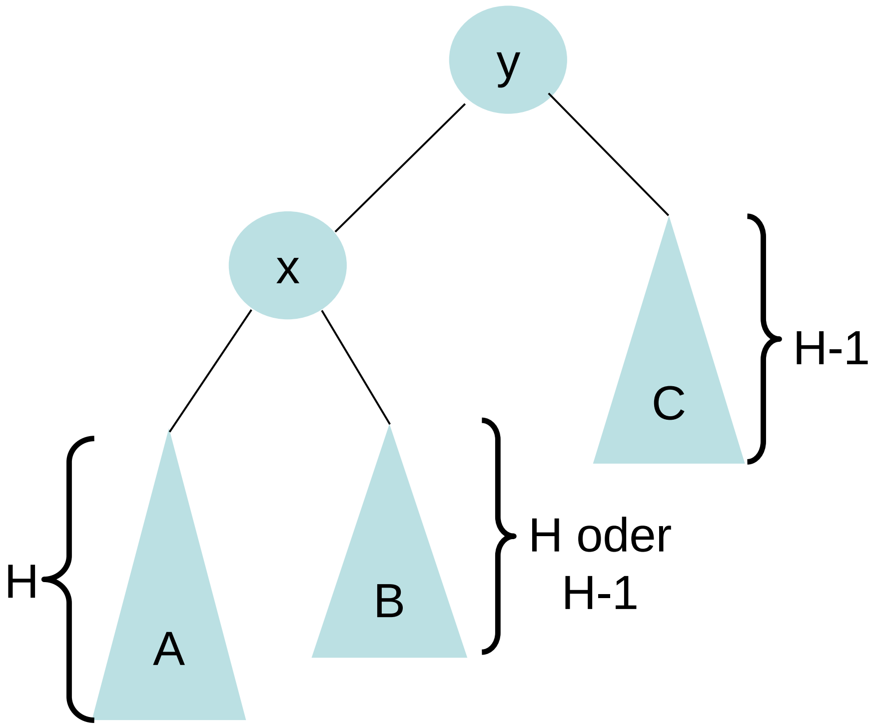


Balancierte binäre Suchbäume

Annahme: beinahe-AVL-Baum mit t=root[T]

Balance(T,t)

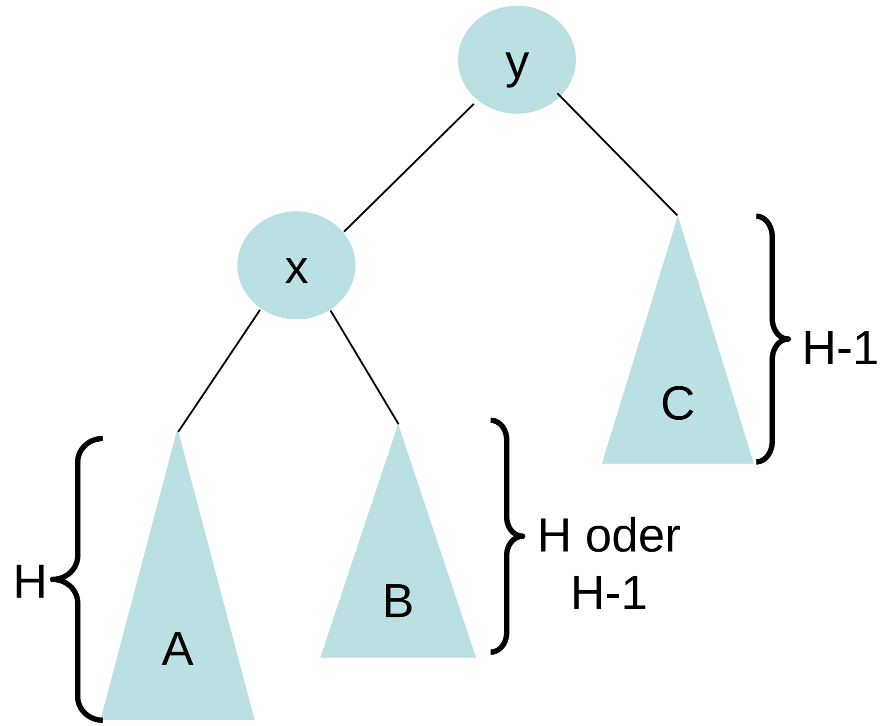
1. **if** h[lc[t]] > h[rc[t]]+1 **then**
2. **if** h[lc[lc[t]]]< h[rc[lc[t]]] **then**
3. Linksrotation(T,lc[t])
4. Rechtsrotation(T,t)
5. **else if** h[rc[t]]> h[lc[t]]+1 **then**
6. **if** h[rc[rc[t]]]< h[lc[rc[t]] **then**
7. Rechtsrotation(T,rc[t])
8. Linksrotation(T,t)



Balancierte binäre Suchbäume

h gibt Höhe des Teilbaums an. Dies müssen wir zusätzlich in unserer Datenstruktur aufrecht erhalten.
Zeiger z undefiniert oder nil: h[z]:=-1

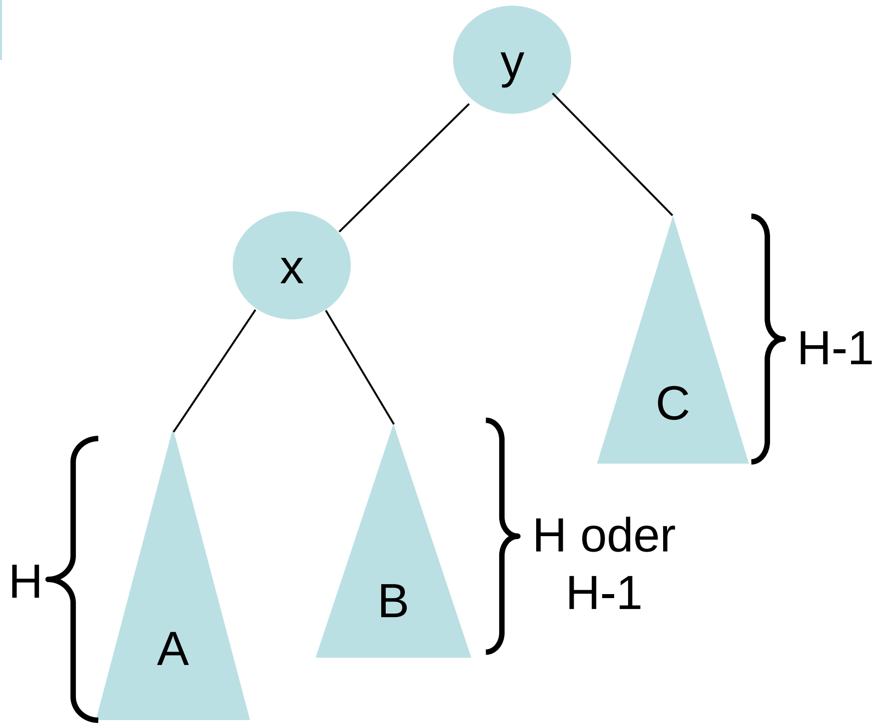
Balance(T,t)
1. **if** h[lc[t]] > h[rc[t]]+1 **then**
2.   **if** h[lc[lc[t]]]< h[rc[lc[t]]] **then**
3.   Linksrotation(T,lc[t])
4.   Rechtsrotation(T,t)
5. **else if** h[rc[t]]> h[lc[t]]+1 **then**
6.   **if** h[rc[rc[t]]]< h[lc[rc[t]] **then**
7.   Rechtsrotation(T,rc[t])
8.   Linksrotation(T,t)



Balancierte binäre Suchbäume

Balance(T,t)

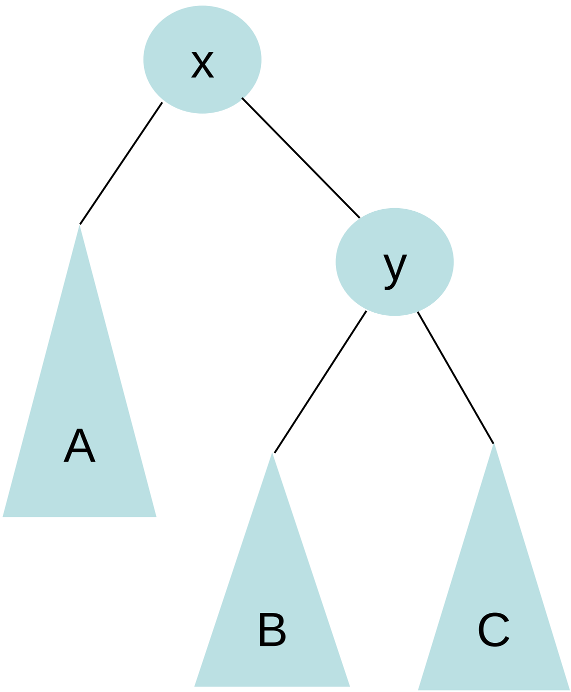
1. **if** h[lc[t]] > h[rc[t]]+1 **then**
2. **if** h[lc[lc[t]]]< h[rc[lc[t]]] **then**
3. Linksrotation(T,lc[t])
4. Rechtsrotation(T,t)
5. **else if** h[rc[t]]> h[lc[t]]+1 **then**
6. **if** h[rc[rc[t]]]< h[lc[rc[t]] **then**
7. Rechtsrotation(T,rc[t])
8. Linksrotation(T,t)



Balancierte binäre Suchbäume

Balance(T,t)

1. **if** h[lc[t]] > h[rc[t]]+1 **then**
2. **if** h[lc[lc[t]]]< h[rc[lc[t]]] **then**
3. Linksrotation(T,lc[t])
4. Rechtsrotation(T,t)
5. **else if** h[rc[t]]> h[lc[t]]+1 **then**
6. **if** h[rc[rc[t]]]< h[lc[rc[t]] **then**
7. Rechtsrotation(T,rc[t])
8. Linksrotation(T,t)

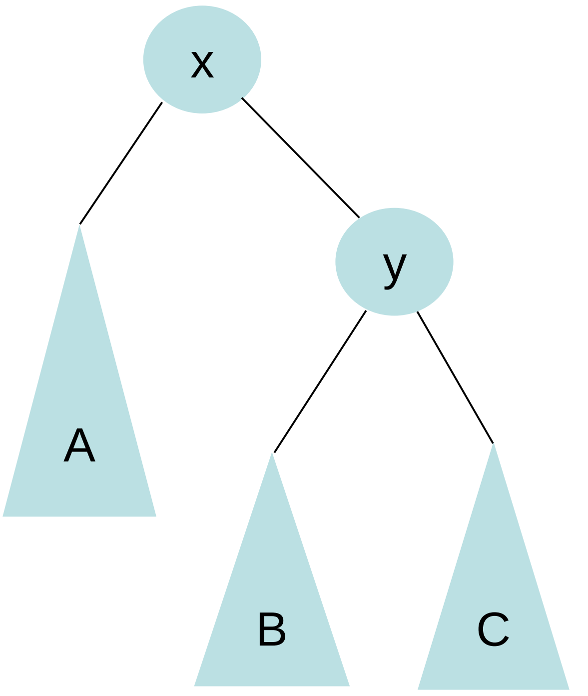


Balancierte binäre Suchbäume

Balance(T,t)

if h[lc[t]] > h[rc[t]]+1 **then**
    if h[lc[lc[t]]]< h[rc[lc[t]]] **then**
        Linksrotation(T,lc[t])
    Rechtsrotation(T,t)
else if h[rc[t]]> h[lc[t]]+1 **then**
    if h[rc[rc[t]]]< h[lc[rc[t]] **then**
        Rechtsrotation(T,rc[t])
    Linksrotation(T,t)

Laufzeit: O(1)



Balancierte binäre Suchbäume

Kurze Zusammenfassung:

- Wir können aus einem beinahe-AVL-Baum mit Hilfe von maximal 2 Rotationen einen AVL-Baum machen
- Dabei bleibt die Höhe des Baums gleich oder nimmt um 1 ab (siehe Schaubilder auf Folien 26-32)

Einfügen:

- Wir fügen ein wie früher
- Dann laufen wir den Pfad zur Wurzel zurück
- An jedem Knoten balancieren wir, falls der Unterbaum ein beinahe-AVL-Baum ist.
- Das ergibt induktiv wieder einen korrekten AVL-Baum.

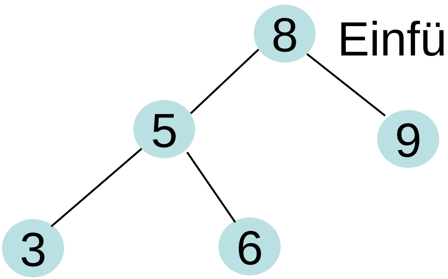
Balancierte binäre Suchbäume

Aufruf mit AVL-Einfügen(nil,root[T],x)

AVL-Einfügen(p,t,x)

1. **if** t=**nil then**
2. füge x an Position t unter p ein; **return**
3. **else if** key[x]<key[t] **then** AVL-Einfügen(t,lc[t],x)
4. **else if** key[x]>key[t] **then** AVL-Einfügen(t,rc[t],x)
5. **else return** ➤ Schlüssel schon vorhanden
6. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
7. Balance(T,t)

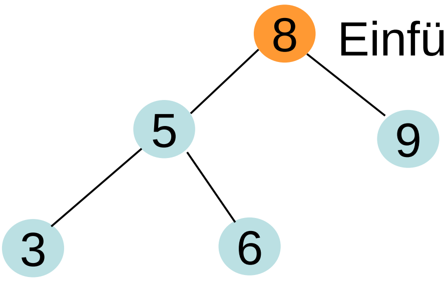
Einfügen 2



Balancierte binäre Suchbäume

AVL-Einfügen(p,t,x)

1. **if** t=**nil then**
2.   füge x an Position t unter p ein; **return**
3. **else if** key[x]<key[t] **then** AVL-Einfügen(t,lc[t],x)
4. **else if** key[x]>key[t] **then** AVL-Einfügen(t,rc[t],x)
5. **else return**   ➢ Schlüssel schon vorhanden
6. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
7. Balance(T,t)

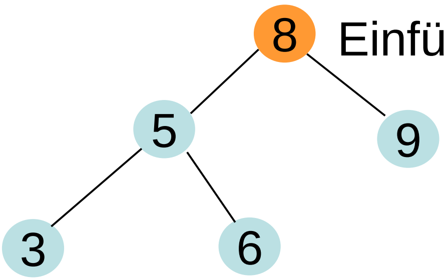


Einfügen 2

Balancierte binäre Suchbäume

AVL-Einfügen(p,t,x)

1. **if** t=**nil then**
2.   füge x an Position t unter p ein; **return**
3. **else if** key[x]<key[t] **then** AVL-Einfügen(t,lc[t],x)
4. **else if** key[x]>key[t] **then** AVL-Einfügen(t,rc[t],x)
5. **else return** ➤ Schlüssel schon vorhanden
6. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
7. Balance(T,t)

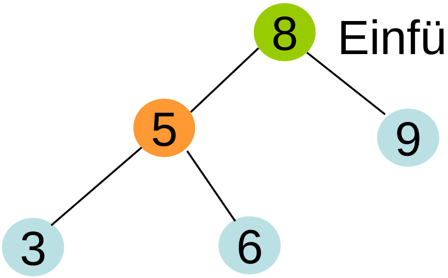


Einfügen 2

Balancierte binäre Suchbäume

AVL-Einfügen(p,t,x)

1. **if** t=**nil then**
2.   füge x an Position t unter p ein; **return**
3. **else if** key[x]<key[t] **then** AVL-Einfügen(t,lc[t],x)
4. **else if** key[x]>key[t] **then** AVL-Einfügen(t,rc[t],x)
5. **else return**   ➢ Schlüssel schon vorhanden
6. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
7. Balance(T,t)

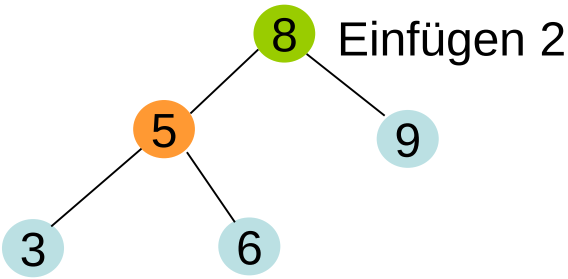


Einfügen 2

Balancierte binäre Suchbäume

AVL-Einfügen(p,t,x)

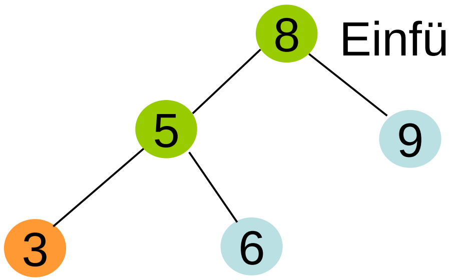
1. **if** t=**nil then**
2. füge x an Position t unter p ein; **return**
3. **else if** key[x]<key[t] **then** AVL-Einfügen(t,lc[t],x)
4. **else if** key[x]>key[t] **then** AVL-Einfügen(t,rc[t],x)
5. **else return** ➢ Schlüssel schon vorhanden
6. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
7. Balance(T,t)



Balancierte binäre Suchbäume

AVL-Einfügen(p,t,x)

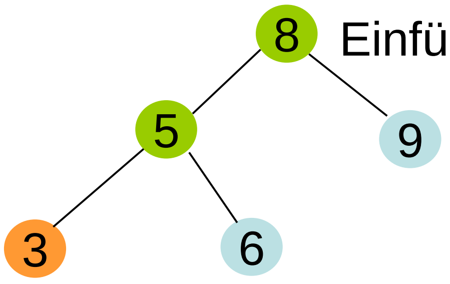
1. **if** t=**nil then**
2. füge x an Position t unter p ein; **return**
3. **else if** key[x]<key[t] **then** AVL-Einfügen(t,lc[t],x)
4. **else if** key[x]>key[t] **then** AVL-Einfügen(t,rc[t],x)
5. **else return** ➢ Schlüssel schon vorhanden
6. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
7. Balance(T,t)



Balancierte binäre Suchbäume

AVL-Einfügen(p,t,x)

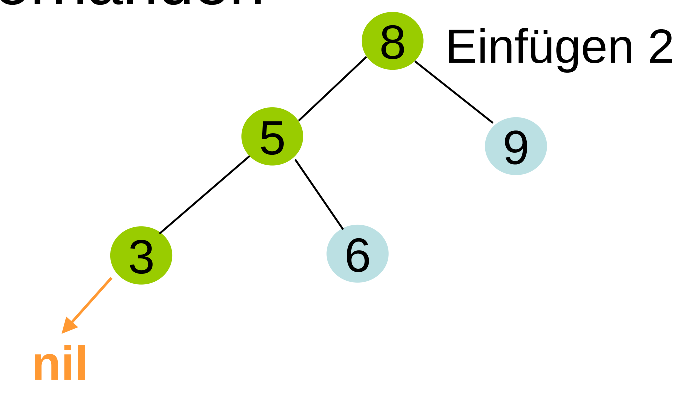
1. **if** t=**nil then**
2.   füge x an Position t unter p ein; **return**
3. **else if** key[x]<key[t] **then** AVL-Einfügen(t,lc[t],x)
4. **else if** key[x]>key[t] **then** AVL-Einfügen(t,rc[t],x)
5. **else return** ➢ Schlüssel schon vorhanden
6. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
7. Balance(T,t)



Balancierte binäre Suchbäume

AVL-Einfügen(p,t,x)

1. **if** t=**nil then**
2. füge x an Position t unter p ein; **return**
3. **else if** key[x]<key[t] **then** AVL-Einfügen(t,lc[t],x)
4. **else if** key[x]>key[t] **then** AVL-Einfügen(t,rc[t],x)
5. **else return** ➢ Schlüssel schon vorhanden
6. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
7. Balance(T,t)

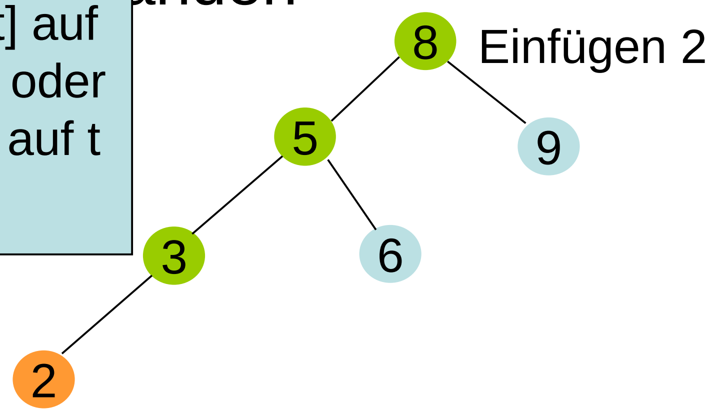


Balancierte binäre Suchbäume

AVL-Einfügen(p,t,x)

1. **if** t=**nil then**
2. füge x an Position t unter p ein; **return**
3. **else if** key[x]<key[t] **then** AVL-Einfügen(t,lc[t],x)
4. **else if** key[x]>key ... n(t,rc[t],x)
5. **else return** ➢ S ... anden
6. h[t] = 1 + max{h[lc ...
7. Balance(T,t)

Neuen Knoten erzeugen, Zeiger lc[t] und rc[t] auf **nil** setzen, sowie p[t] auf p und den Zeiger (lc oder rc) von p (falls ≠ nil) auf t setzen.



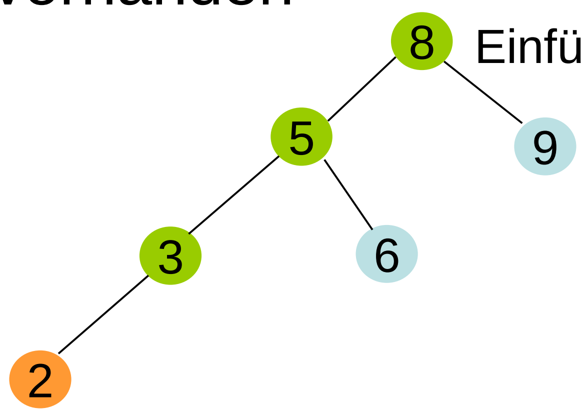
Balancierte binäre Suchbäume

AVL-Einfügen(p,t,x)

1. **if** t=**nil then**
2. füge x an Position t unter p ein; **return**
3. **else if** key[x]<key[t] **then** AVL-Einfügen(t,lc[t],x)
4. **else if** key[x]>key[t] **then** AV
5. **else return** ➤ Schlüssel schon vorhanden
6. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
7. Balance(T,t)

h[**nil**] definieren wir als -1.

Einfügen 2

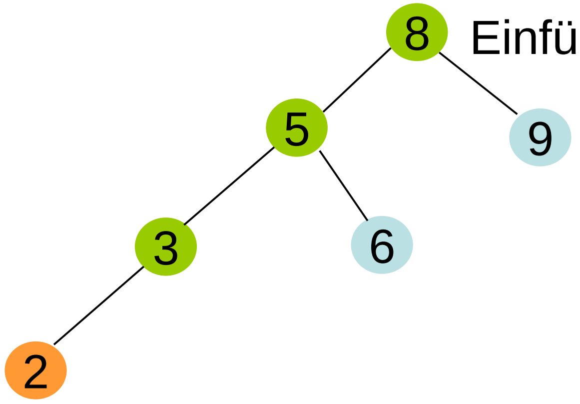


Balancierte binäre Suchbäume

AVL-Einfügen(p,t,x)

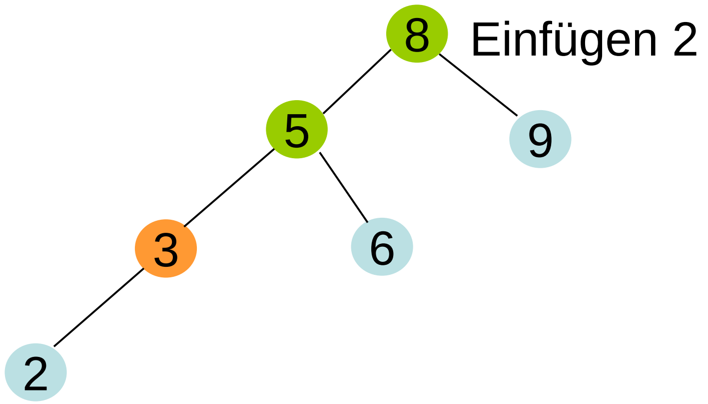
1. **if** t=**nil then**
2. füge x an Position t unter p ein; **return**
3. **else if** key[x]<key[t] **then** AVL-Einfügen(t,lc[t],x)
4. **else if** key[x]>key[t] **then** AVL-Einfügen(t,rc[t],x)
5. **else return** ➢ Schlüssel schon vorhanden
6. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
7. Balance(T,t)

Einfügen 2



Balancierte binäre Suchbäume

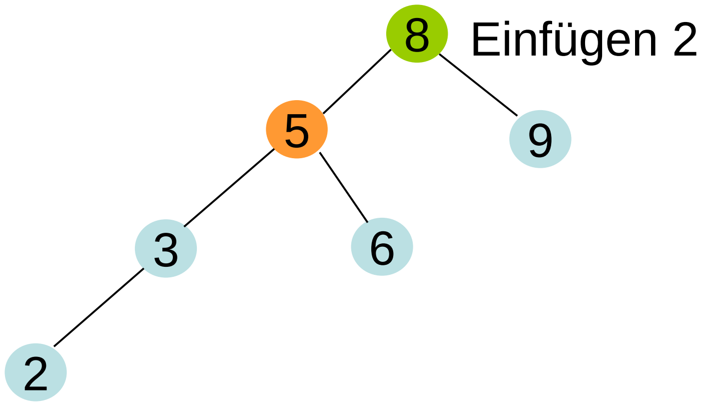
```
AVL-Einfügen(p,t,x)
if t=nil then
    füge x an Position t unter p ein; return
else if key[x]<key[t] then AVL-Einfügen(t,lc[t],x)
else if key[x]>key[t] then AVL-Einfügen(t,rc[t],x)
else return     ➢ Schlüssel schon vorhanden
h[t] = 1 + max{h[lc[t]], h[rc[t]]}
Balance(T,t)
```



Balancierte binäre Suchbäume

AVL-Einfügen(p,t,x)

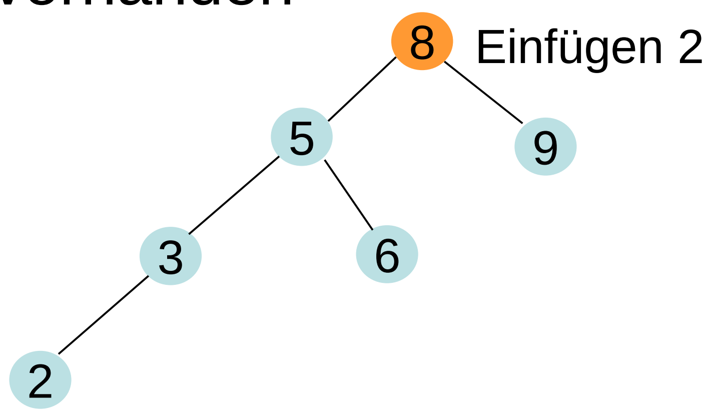
1. **if** t=**nil then**
2.   füge x an Position t unter p ein; **return**
3. **else if** key[x]<key[t] **then** AVL-Einfügen(t,lc[t],x)
4. **else if** key[x]>key[t] **then** AVL-Einfügen(t,rc[t],x)
5. **else return**   ➢ Schlüssel schon vorhanden
6. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
7. Balance(T,t)



Balancierte binäre Suchbäume

AVL-Einfügen(p,t,x)

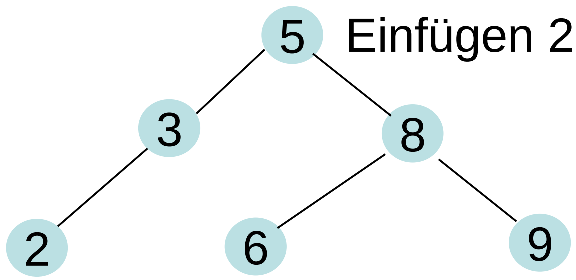
1. **if** t=**nil then**
2.   füge x an Position t unter p ein; **return**
3. **else if** key[x]<key[t] **then** AVL-Einfügen(t,lc[t],x)
4. **else if** key[x]>key[t] **then** AVL-Einfügen(t,rc[t],x)
5. **else return**   ➢ Schlüssel schon vorhanden
6. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
7. Balance(T,t)



Balancierte binäre Suchbäume

AVL-Einfügen(p,t,x)

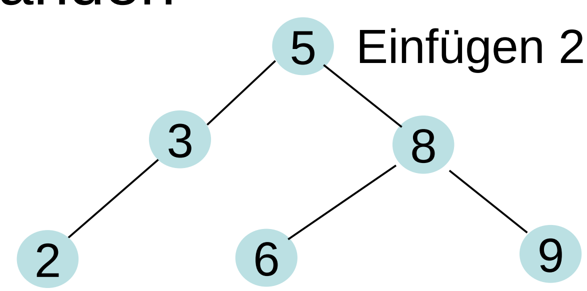
1. **if** t=**nil then**
2.   füge x an Position t unter p ein; **return**
3. **else if** key[x]<key[t] **then** AVL-Einfügen(t,lc[t],x)
4. **else if** key[x]>key[t] **then** AVL-Einfügen(t,rc[t],x)
5. **else return**   ➢ Schlüssel schon vorhanden
6. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
7. Balance(T,t)



Balancierte binäre Suchbäume

AVL-Einfügen(p,t,x)

1. **if** t=**nil then**
2. füge x an Position t unter p ein; **return**
3. **else if** key[x]<key[t] **then** AVL-Einfügen(t,lc[t],x)
4. **else if** key[x]>key[t] **then** AVL-Einfügen(t,rc[t],x)
5. **else return** ➢ Schlüssel schon vorhanden
6. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
7. Balance(T,t)



Laufzeit:

- O(h) = O(log n)

Balancierte binäre Suchbäume

Korrektheit:

- $h(v)$: Höhe das Baumes mit Wurzel v vor dem Einfügen von x
- $h´(v)$: Höhe das Baumes mit Wurzel v nach dem Einfügen von x

Direkt nach dem Einfügen von x gilt:

- Für alle Knoten v entlang des Suchpfades von x, $h´(v) \in \{h(v), h(v)+1\}$, und für alle anderen ist $h´(v)=h(v)$.
- Ist $h´(v)=h(v)+1$, dann ist auch $h´(w)=h(w)+1$ für das Kind w von v in Richtung x (da der andere Teilbaum von v die Höhe beibehalten hat).
- Da $h´(x)=h(x)+1$ (an der Position von x war vorher nil, und $h(nil)=-1$), gibt es einen Vorfahren v von x mit $h´(w)=h(w)+1$ für alle Knoten w entlang des Suchpfades von v nach x, und für alle anderen Knoten w gilt, $h´(w)=h(w)$.

Balancierte binäre Suchbäume

Korrektheit:

- h(v): Höhe das Baumes mit Wurzel v vor dem Einfügen
- h´(v): Höhe das Baumes mit Wurzel v nach dem Einfügen

Wir wissen:

- Bei einer Balancierung bleibt die Höhe gleich oder sinkt um 1.
- Es gibt einen Vorfahren v von x mit h´(w)=h(w)+1 für alle Knoten w entlang des Suchpfades von v nach x, und für alle anderen Knoten w gilt, h´(w)=h(w).
- Eine Balancierung kann bei w nur dann stattfinden, wenn h´(w)=h(w)+1 ist (da sonst die AVL-Eigenschaft gilt).
- Sobald zum ersten Mal die Balancierung eines Vorfahrens w von x ergibt, dass danach h´(w)=h(w) ist, dann korrigieren sich dadurch auch die Höhen aller anderen Vorfahren y von x über w auf h´(y)=h(y), so dass keine weiteren Balancierungen mehr notwendig sind.

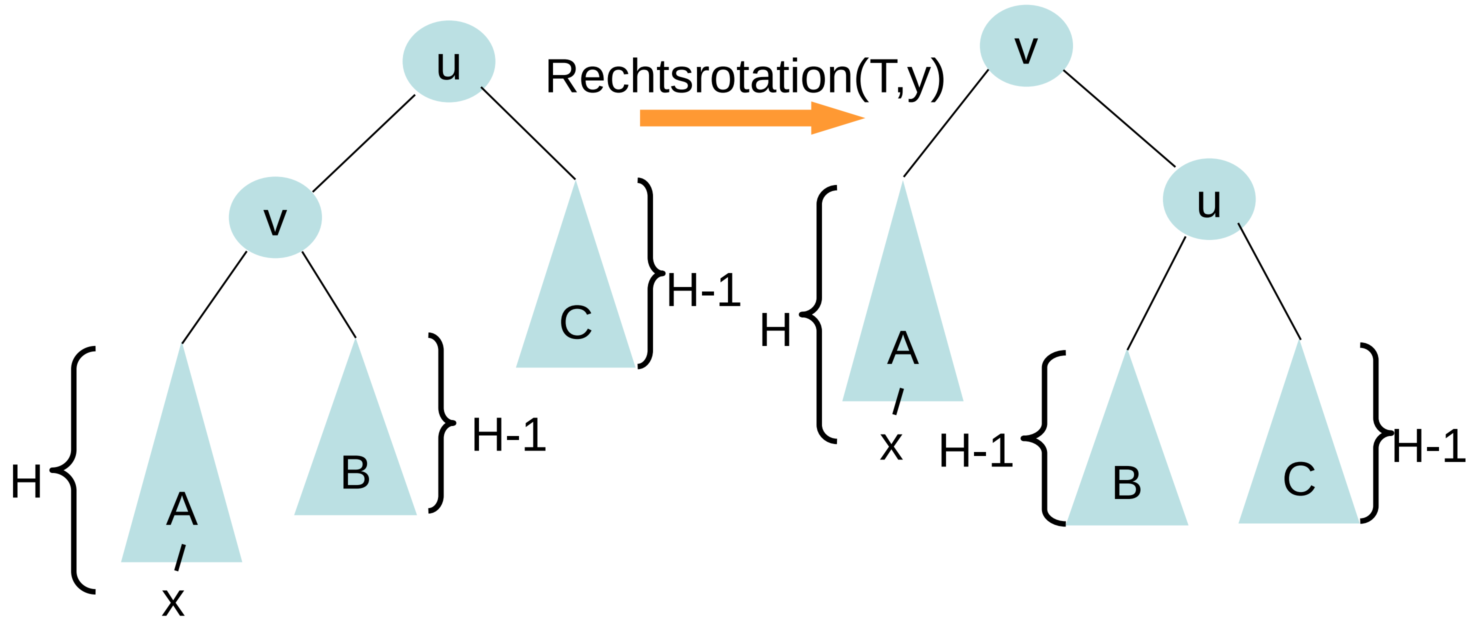
Balancierte binäre Suchbäume

Korrektheit:

Behauptung: Eine Rebalancierung ist nur genau einmal notwendig.

Beweis:

- Fall 1: einfache Rotation.
  Da h´(u)=h(u)+1, muss Höhe von Teilbaum A angewachsen sein,
  d.h. für die aktuellen Höhen muss für ein H gelten:



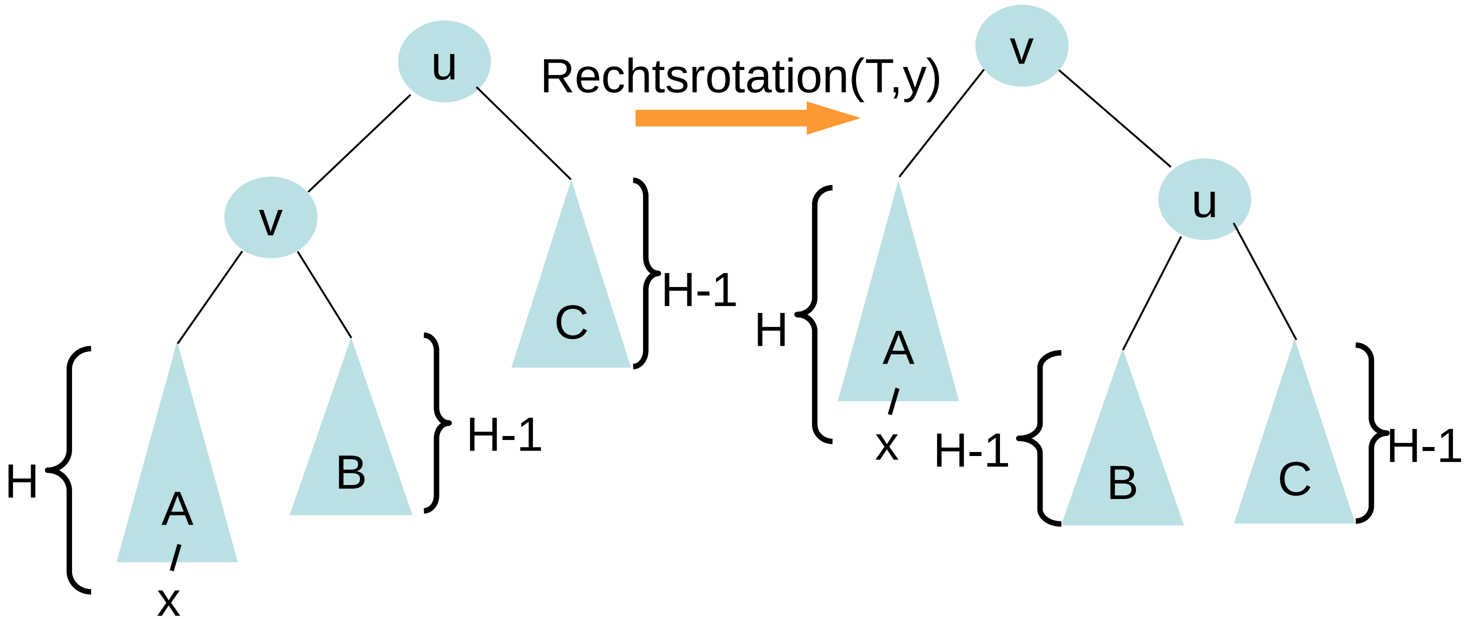
Balancierte binäre Suchbäume

Korrektheit:

Behauptung: Eine Rebalancierung ist nur genau einmal notwendig.

Beweis:

- Fall 1: einfache Rotation.
  Da nachher h´(v)=h(u) ist, gibt es keine weiteren Rebalancierungen mehr.



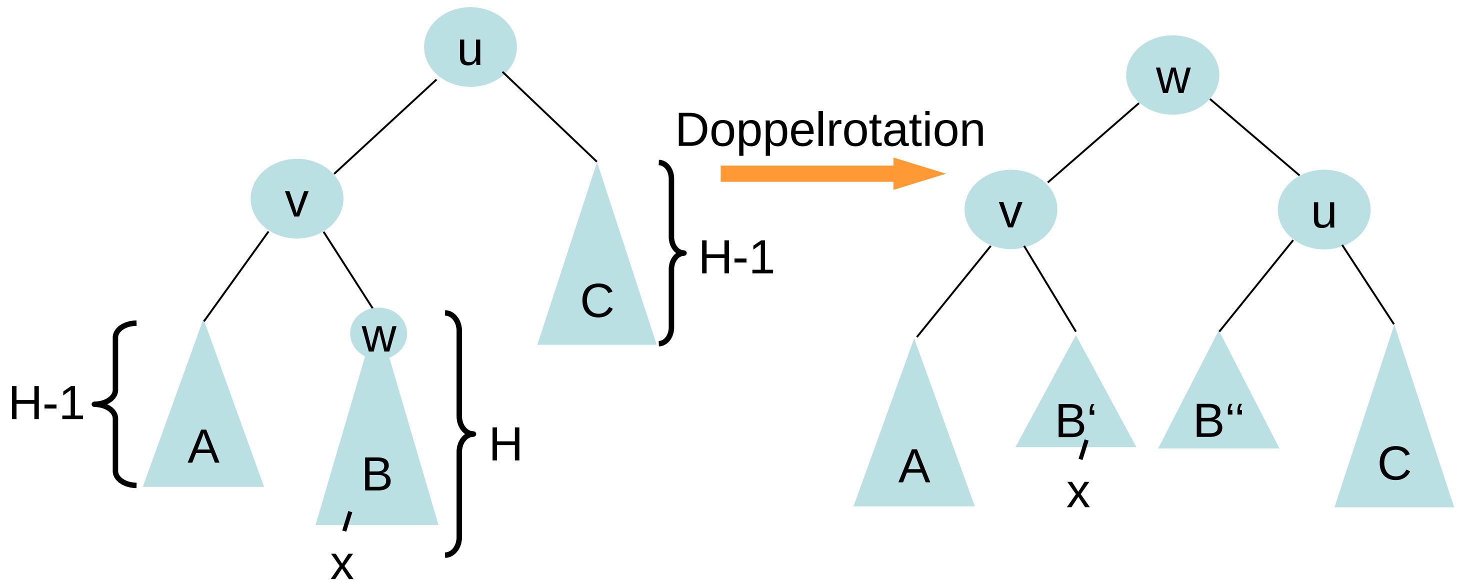
Balancierte binäre Suchbäume

Korrektheit:

Behauptung: Eine Rebalancierung ist nur genau einmal notwendig.

Beweis:

- Fall 2: Doppelrotation.
  Da h´(u)=h(u)+1, muss Höhe von Teilbaum B angewachsen sein, d.h. für die aktuellen Höhen muss für ein H gelten:



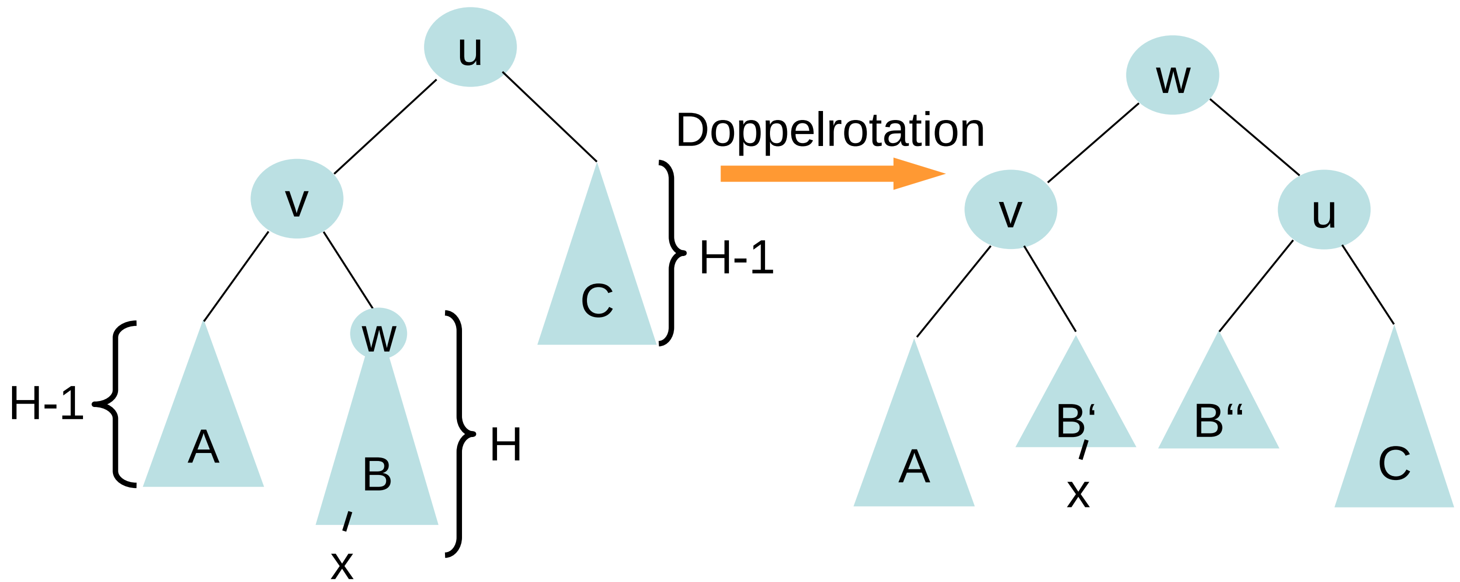
Balancierte binäre Suchbäume

Korrektheit:

Behauptung: Eine Rebalancierung ist nur genau einmal notwendig.

Beweis:

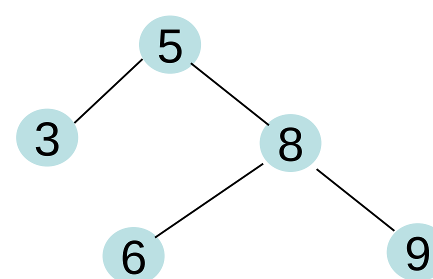
- Fall 2: Doppelrotation.
  Da nachher h´(w)=h(u) ist, gibt es auch hier keine weiteren Rebalancierungen mehr.



Balancierte binäre Suchbäume

AVL-Löschen(t,k)

if k<key[t] **then** AVL-Löschen(lc[t],k)
e**lse** if k>key **then** AVL-Löschen(rc[t],k)
e**lse if** t=**nil then return** ➢ k nicht im Baum
else if lc[t]=**nil then** ersetze t durch rc[t]
else if rc[t]=**nil then** ersetze t durch lc[t]
e**lse** u=MaximumSuche(lc[t])
    Kopiere Informationen von u nach t
    AVL-Löschen(lc[t],key[u])
h[t] = 1 + max{h[lc[t]], h[rc[t]]}
Balance(T,t)



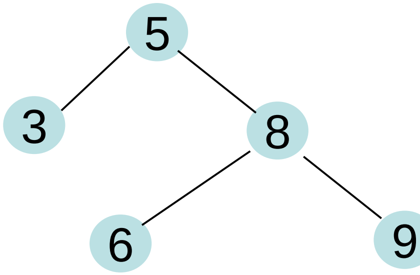
Balancierte binäre Suchbäume

k bezeichnet Schlüssel des zu löschenden Elements.

AVL-Löschen(t,k)

1. **if** k<key[t] **then** AVL-Löschen(lc[t],k)
2. e**lse** if k>key **then** AVL-Löschen(rc[t],k)
3. e**lse if** t=**nil then return** ➢ k nicht im Baum
4. **else if** lc[t]=**nil then** ersetze t durch rc[t]
5. **else if** rc[t]=**nil then** ersetze t durch lc[t]
6. e**lse** u=MaximumSuche(lc[t])
7. Kopiere Informationen von u nach t
8. AVL-Löschen(lc[t],key[u])
9. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
10. Balance(T,t)

Löschen(3)

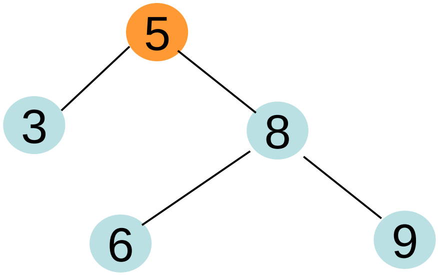


Balancierte binäre Suchbäume

AVL-Löschen(t,k)

1. **if** k<key[t] **then** AVL-Löschen(lc[t],k)
2. e**lse** if k>key **then** AVL-Löschen(rc[t],k)
3. e**lse if** t=**nil then return** ➢ k nicht im Baum
4. **else if** lc[t]=**nil then** ersetze t durch rc[t]
5. **else if** rc[t]=**nil then** ersetze t durch lc[t]
6. e**lse** u=MaximumSuche(lc[t])
7. Kopiere Informationen von u nach t
8. AVL-Löschen(lc[t],key[u])
9. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
10. Balance(T,t)

Löschen(3)

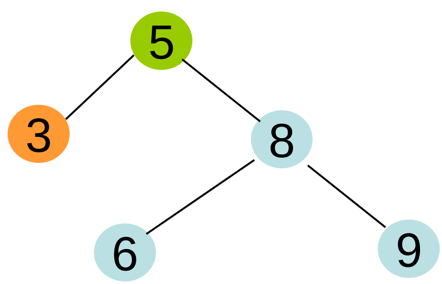


Balancierte binäre Suchbäume

AVL-Löschen(t,k)

1. **if** k<key[t] **then** AVL-Löschen(lc[t],k)
2. e**lse** if k>key **then** AVL-Löschen(rc[t],k)
3. e**lse if** t=**nil then return** ➢ k nicht im Baum
4. **else if** lc[t]=**nil then** ersetze t durch rc[t]
5. **else if** rc[t]=**nil then** ersetze t durch lc[t]
6. e**lse** u=MaximumSuche(lc[t])
7. Kopiere Informationen von u nach t
8. AVL-Löschen(lc[t],key[u])
9. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
10. Balance(T,t)

Löschen(3)



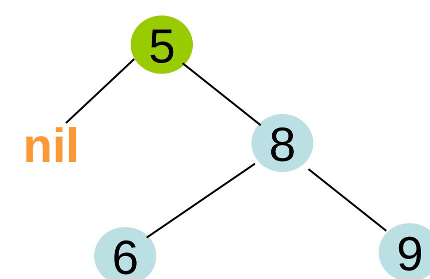
Balancierte binäre Suchbäume

AVL-Löschen(t,k)

1. **if** k<key[t] **then** AVL-Löschen(lc[t],k)
2. e**lse** if k>key **then** AVL-Löschen(rc
3. e**lse if** t=**nil then return** ➢ k nicht im Baum
4. **else if** lc[t]=**nil then** ersetze t durch rc[t]
5. **else if** rc[t]=**nil then** ersetze t durch lc[t]
6. e**lse** u=MaximumSuche(lc[t])
7. Kopiere Informationen von u nach t
8. AVL-Löschen(lc[t],key[u])
9. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
10. Balance(T,t)

Und die anderen Zeiger aktualisieren

Löschen(3)



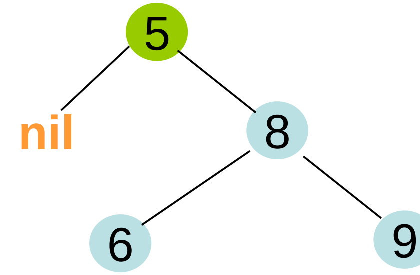
Balancierte binäre Suchbäume

AVL-Löschen(t,k)

1. **if** k<key[t] **then** AVL-Löschen(lc[t],k)
2. e**lse** if k>key **then** AVL-Löschen(rc[t],k)
3. e**lse if** t=**nil then return** ➢ k nicht im Baum
4. **else if** lc[t]=**nil then** ersetze t durch rc[t]
5. **else if** rc[t]=**nil then** ersetze t durch lc[t]
6. e**lse** u=MaximumSuche(lc[t])
7. Kopiere Informationen vo
8. AVL-Löschen(lc[t],key[u])
9. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
10. Balance(T,t)

Oder h[t]=-1, falls t=nil

Löschen(3)



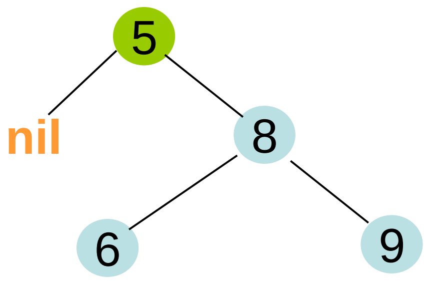
Balancierte binäre Suchbäume

AVL-Löschen(t,k)

1. **if** k<key[t] **then** AVL-Löschen(lc[t],k)
2. e**lse** if k>key **then** AVL-Löschen(rc[t],k)
3. e**lse if** t=**nil then return** ➢ k nicht im Baum
4. **else if** lc[t]=**nil then** ersetze t durch rc[t]
5. **else if** rc[t]=**nil then** ersetze t durch lc[t]
6. e**lse** u=MaximumSuche(lc[t])
7. Kopiere Informationen von u nach t
8. AVL-Lös ... y[u])
9. h[t] = 1 + max ... [t]]}
10. Balance(T,t)

Nichts zu tun,
da Baum leer.

Löschen(3)

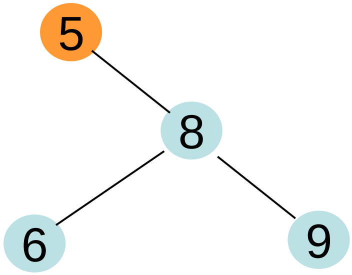


Balancierte binäre Suchbäume

AVL-Löschen(t,k)

1. **if** k<key[t] **then** AVL-Löschen(lc[t],k)
2. e**lse** if k>key **then** AVL-Löschen(rc[t],k)
3. e**lse if** t=**nil then return** ➢ k nicht im Baum
4. **else if** lc[t]=**nil then** ersetze t durch rc[t]
5. **else if** rc[t]=**nil then** ersetze t durch lc[t]
6. e**lse** u=MaximumSuche(lc[t])
7. Kopiere Informationen von u nach t
8. AVL-Löschen(lc[t],key[u])
9. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
10. Balance(T,t)

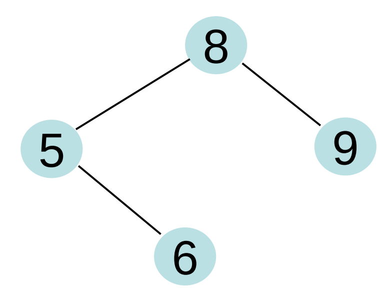
Löschen(3)



Balancierte binäre Suchbäume

```
AVL-Löschen(t,k)
if k<key[t] then AVL-Löschen(lc[t],k)
else if k>key then AVL-Löschen(rc[t],k)
else if t=nil then return ➢ k nicht im Baum
else if lc[t]=nil then ersetze t durch rc[t]
else if rc[t]=nil then ersetze t durch lc[t]
else u=MaximumSuche(lc[t])
        Kopiere Informationen von u nach t
        AVL-Löschen(lc[t],key[u])
h[t] = 1 + max{h[lc[t]], h[rc[t]]}
Balance(T,t)
```

Löschen(3)

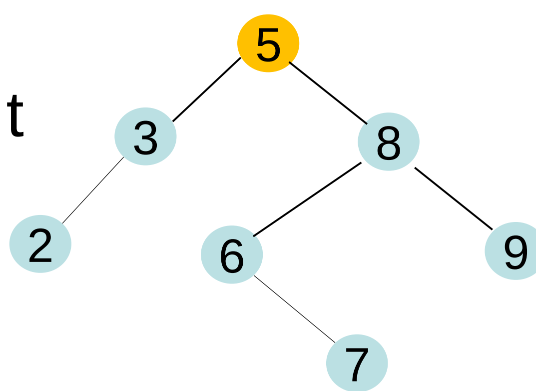


Balancierte binäre Suchbäume

AVL-Löschen(t,k)

1. **if** k<key[t] **then** AVL-Löschen(lc[t],k)
2. e**lse** if k>key **then** AVL-Löschen(rc[t],k)
3. e**lse if** t=**nil then return** ➢ k nicht im Baum
4. **else if** lc[t]=**nil then** ersetze t durch rc[t]
5. **else if** rc[t]=**nil then** ersetze t durch lc[t]
6. e**lse** u=MaximumSuche(lc[t])
7. Kopiere Informationen von u nach t
8. AVL-Löschen(lc[t],key[u])
9. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
10. Balance(T,t)

Löschen(8)

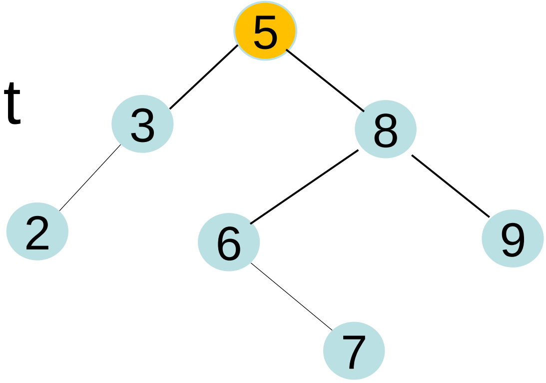


Balancierte binäre Suchbäume

AVL-Löschen(t,k)

1. **if** k<key[t] **then** AVL-Löschen(lc[t],k)
2. e**lse** if k>key **then** AVL-Löschen(rc[t],k)
3. e**lse if** t=**nil then return** ➢ k nicht im Baum
4. **else if** lc[t]=**nil then** ersetze t durch rc[t]
5. **else if** rc[t]=**nil then** ersetze t durch lc[t]
6. e**lse** u=MaximumSuche(lc[t])
7. Kopiere Informationen von u nach t
8. AVL-Löschen(lc[t],key[u])
9. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
10. Balance(T,t)

Löschen(8)

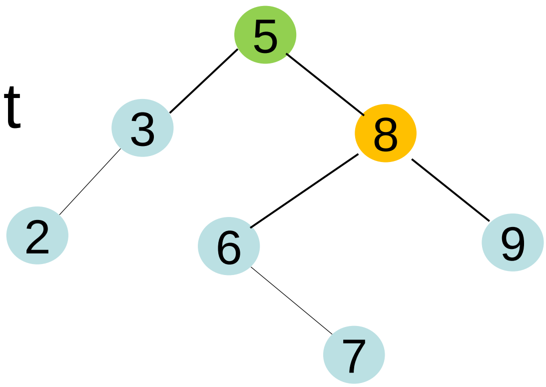


Balancierte binäre Suchbäume

AVL-Löschen(t,k)

1. **if** k<key[t] **then** AVL-Löschen(lc[t],k)
2. e**lse** if k>key **then** AVL-Löschen(rc[t],k)
3. e**lse if** t=**nil then return** ➢ k nicht im Baum
4. **else if** lc[t]=**nil then** ersetze t durch rc[t]
5. **else if** rc[t]=**nil then** ersetze t durch lc[t]
6. e**lse** u=MaximumSuche(lc[t])
7. Kopiere Informationen von u nach t
8. AVL-Löschen(lc[t],key[u])
9. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
10. Balance(T,t)

Löschen(8)

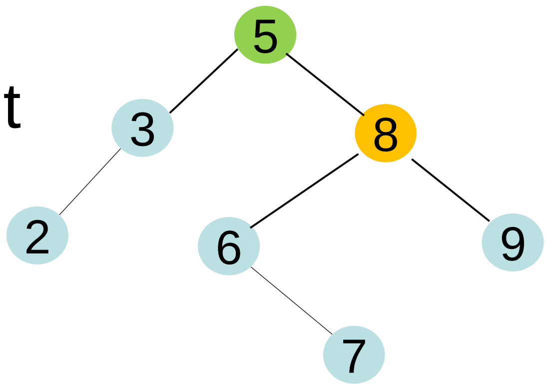


Balancierte binäre Suchbäume

AVL-Löschen(t,k)

1. **if** k<key[t] **then** AVL-Löschen(lc[t],k)
2. e**lse** if k>key **then** AVL-Löschen(rc[t],k)
3. e**lse if** t=**nil then return** ➢ k nicht im Baum
4. **else if** lc[t]=**nil then** ersetze t durch rc[t]
5. **else if** rc[t]=**nil then** ersetze t durch lc[t]
6. e**lse** u=MaximumSuche(lc[t])
7. Kopiere Informationen von u nach t
8. AVL-Löschen(lc[t],key[u])
9. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
10. Balance(T,t)

Löschen(8)

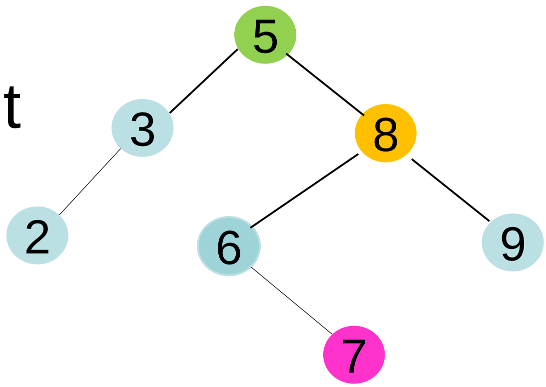


Balancierte binäre Suchbäume

AVL-Löschen(t,k)

1. **if** k<key[t] **then** AVL-Löschen(lc[t],k)
2. e**lse** if k>key **then** AVL-Löschen(rc[t],k)
3. e**lse if** t=**nil then return** ➢ k nicht im Baum
4. **else if** lc[t]=**nil then** ersetze t durch rc[t]
5. **else if** rc[t]=**nil then** ersetze t durch lc[t]
6. e**lse** u=MaximumSuche(lc[t])
7. Kopiere Informationen von u nach t
8. AVL-Löschen(lc[t],key[u])
9. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
10. Balance(T,t)

Löschen(8)

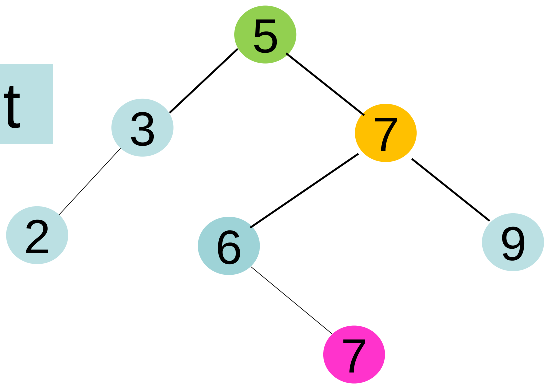


Balancierte binäre Suchbäume

AVL-Löschen(t,k)

1. **if** k<key[t] **then** AVL-Löschen(lc[t],k)
2. e**lse** if k>key **then** AVL-Löschen(rc[t],k)
3. e**lse if** t=**nil then return** ➢ k nicht im Baum
4. **else if** lc[t]=**nil then** ersetze t durch rc[t]
5. **else if** rc[t]=**nil then** ersetze t durch lc[t]
6. e**lse** u=MaximumSuche(lc[t])
7. Kopiere Informationen von u nach t
8. AVL-Löschen(lc[t],key[u])
9. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
10. Balance(T,t)

Löschen(8)



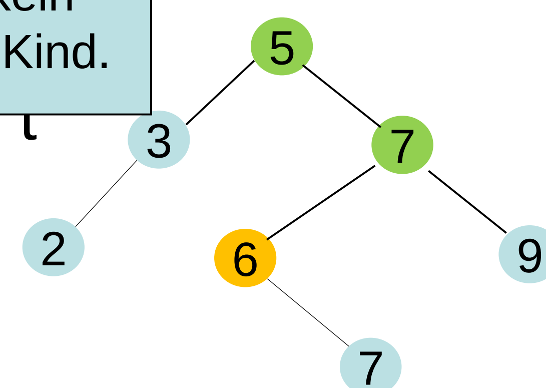
Balancierte binäre Suchbäume

AVL-Löschen(t,k)

1. **if** k<key[t] **then** AVL-Löschen(lc[t],k)
2. e**lse** if k>key **then** AVL-Löschen(rc[t],k)
3. e**lse if** t=**nil then return** ➢ k nicht im Baum
4. **else if** lc[t]=**nil then** ersetze t durch rc[t]
5. **else if** rc[t]=**nil then** ersetze t du
6. e**lse** u=MaximumSuche(lc[t])
7. Kopiere Informationen von u nach t
8. AVL-Löschen(lc[t],key[u])
9. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
10. Balance(T,t)

u hat kein rechtes Kind.

Löschen(8)

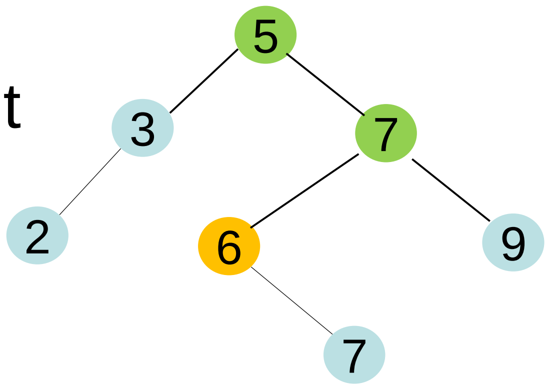


Balancierte binäre Suchbäume

AVL-Löschen(t,k)

1. **if** k<key[t] **then** AVL-Löschen(lc[t],k)
2. e**lse** if k>key **then** AVL-Löschen(rc[t],k)
3. e**lse if** t=**nil then return** ➢ k nicht im Baum
4. **else if** lc[t]=**nil then** ersetze t durch rc[t]
5. **else if** rc[t]=**nil then** ersetze t durch lc[t]
6. e**lse** u=MaximumSuche(lc[t])
7. Kopiere Informationen von u nach t
8. AVL-Löschen(lc[t],key[u])
9. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
10. Balance(T,t)

Löschen(8)

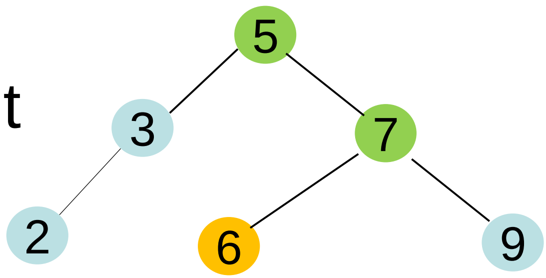


Balancierte binäre Suchbäume

AVL-Löschen(t,k)

1. **if** k<key[t] **then** AVL-Löschen(lc[t],k)
2. e**lse** if k>key **then** AVL-Löschen(rc[t],k)
3. e**lse if** t=**nil then return** ➢ k nicht im Baum
4. **else if** lc[t]=**nil then** ersetze t durch rc[t]
5. **else if** rc[t]=**nil then** ersetze t durch lc[t]
6. e**lse** u=MaximumSuche(lc[t])
7.   Kopiere Informationen von u nach t
8.   AVL-Löschen(lc[t],key[u])
9. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
10. Balance(T,t)

Löschen(8)

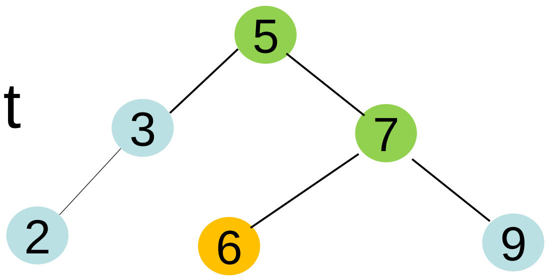


Balancierte binäre Suchbäume

AVL-Löschen(t,k)

1. **if** k<key[t] **then** AVL-Löschen(lc[t],k)
2. **else** if k>key **then** AVL-Löschen(rc[t],k)
3. **else if** t=nil **then return** ➢ k nicht im Baum
4. **else if** lc[t]=**nil then** ersetze t durch rc[t]
5. **else if** rc[t]=**nil then** ersetze t durch lc[t]
6. **else** u=MaximumSuche(lc[t])
7. Kopiere Informationen von u nach t
8. AVL-Löschen(lc[t],key[u])
9. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
10. Balance(T,t)

Löschen(8)

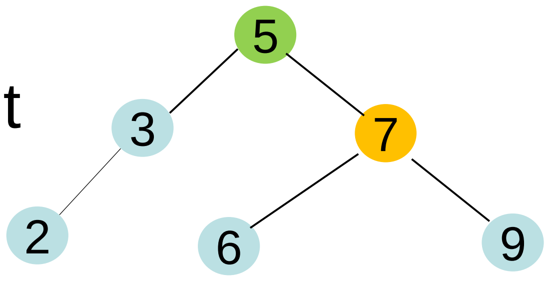


Balancierte binäre Suchbäume

AVL-Löschen(t,k)

1. **if** k<key[t] **then** AVL-Löschen(lc[t],k)
2. **else** if k>key **then** AVL-Löschen(rc[t],k)
3. **else if** t=**nil then return** ➢ k nicht im Baum
4. **else if** lc[t]=**nil then** ersetze t durch rc[t]
5. **else if** rc[t]=**nil then** ersetze t durch lc[t]
6. **else** u=MaximumSuche(lc[t])
7. Kopiere Informationen von u nach t
8. AVL-Löschen(lc[t],key[u])
9. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
10. Balance(T,t)

Löschen(8)

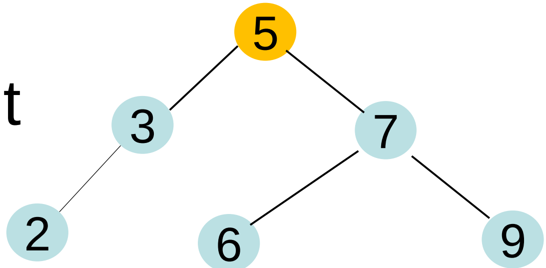


Balancierte binäre Suchbäume

AVL-Löschen(t,k)

1. **if** k<key[t] **then** AVL-Löschen(lc[t],k)
2. **else** if k>key **then** AVL-Löschen(rc[t],k)
3. **else if** t=**nil then return** ➢ k nicht im Baum
4. **else if** lc[t]=**nil then** ersetze t durch rc[t]
5. **else if** rc[t]=**nil then** ersetze t durch lc[t]
6. **else** u=MaximumSuche(lc[t])
7. Kopiere Informationen von u nach t
8. AVL-Löschen(lc[t],key[u])
9. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
10. Balance(T,t)

Löschen(8)

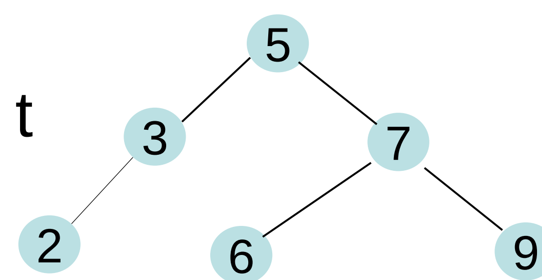


Balancierte binäre Suchbäume

AVL-Löschen(t,k)

1. **if** k<key[t] **then** AVL-Löschen(lc[t],k)
2. **else** if k>key **then** AVL-Löschen(rc[t],k)
3. **else if** t=**nil then return** ➢ k nicht im Baum
4. **else if** lc[t]=**nil then** ersetze t durch rc[t]
5. **else if** rc[t]=**nil then** ersetze t durch lc[t]
6. **else** u=MaximumSuche(lc[t])
7. Kopiere Informationen von u nach t
8. AVL-Löschen(lc[t],key[u])
9. h[t] = 1 + max{h[lc[t]], h[rc[t]]}
10. Balance(T,t)

Löschen(8)



Balancierte binäre Suchbäume

Korrektheit:

- $h(v)$: Höhe das Baumes mit Wurzel v vor dem Einfügen von x
- $h'(v)$: Höhe das Baumes mit Wurzel v nach dem Einfügen von x

Direkt nach dem Löschen von x (oder einem Nachfahren von x, den wir in diesem Fall auch mit x bezeichnen) gilt:

- Für alle Knoten v entlang des Suchpfades von der Wurzel zum Elternknoten y von x gilt $h'(v) \in \{h(v), h(v)-1\}$, und für alle anderen ist $h'(v)=h(v)$.
- Ist $h'(v)=h(v)-1$, dann ist auch $h'(w)=h(w)-1$ für das Kind w von v in Richtung y (da der andere Teilbaum von v die Höhe beibehalten hat).
- Gilt für den Elternknoten y, dass $h'(y)=h(y)$, muss demnach für alle Knoten v gelten, dass $h'(v)=h(v)$, und es ist nichts zu rebalancieren. Wir nehmen daher im folgenden an, dass $h'(y)=h(y)-1$.

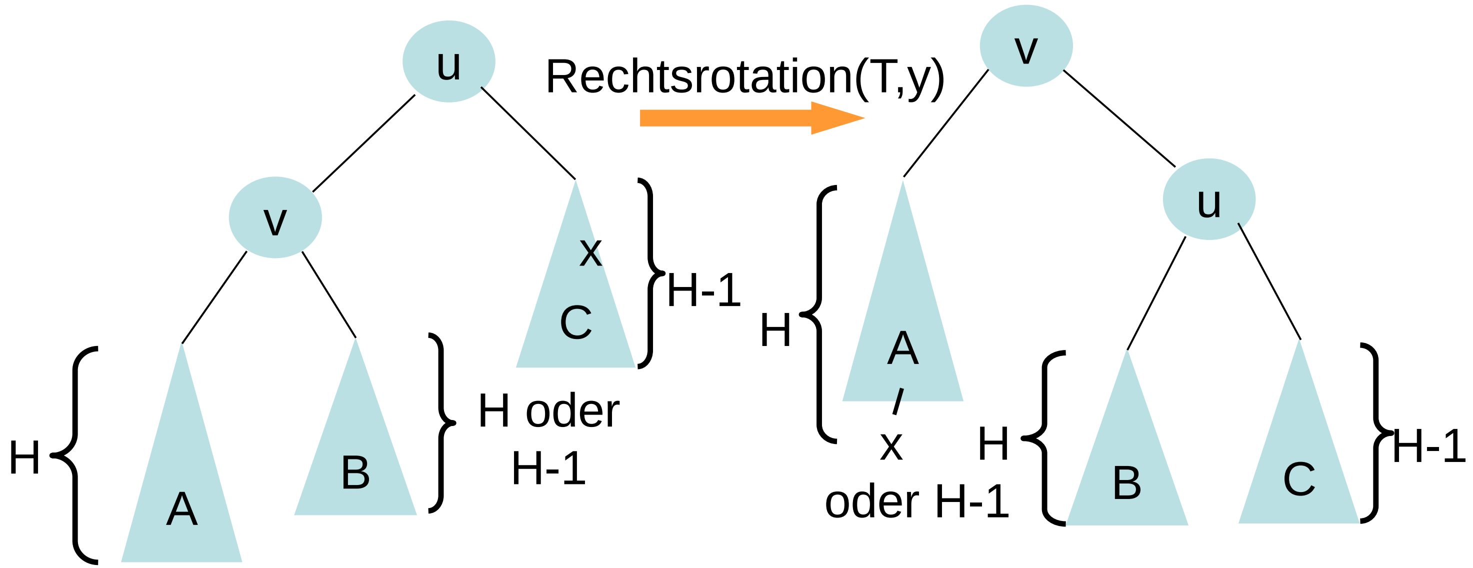
Balancierte binäre Suchbäume

Korrektheit:

Behauptung: Eine Rebalancierung bei u ist nur für h´(u)=h(u) möglich.

Beweis:

- Fall 1: einfache Rotation.
  Kann nur passieren, wenn sich die Höhe von C verkleinert hat. In diesem Fall gilt dann aber, dass h´(u)=h(u).



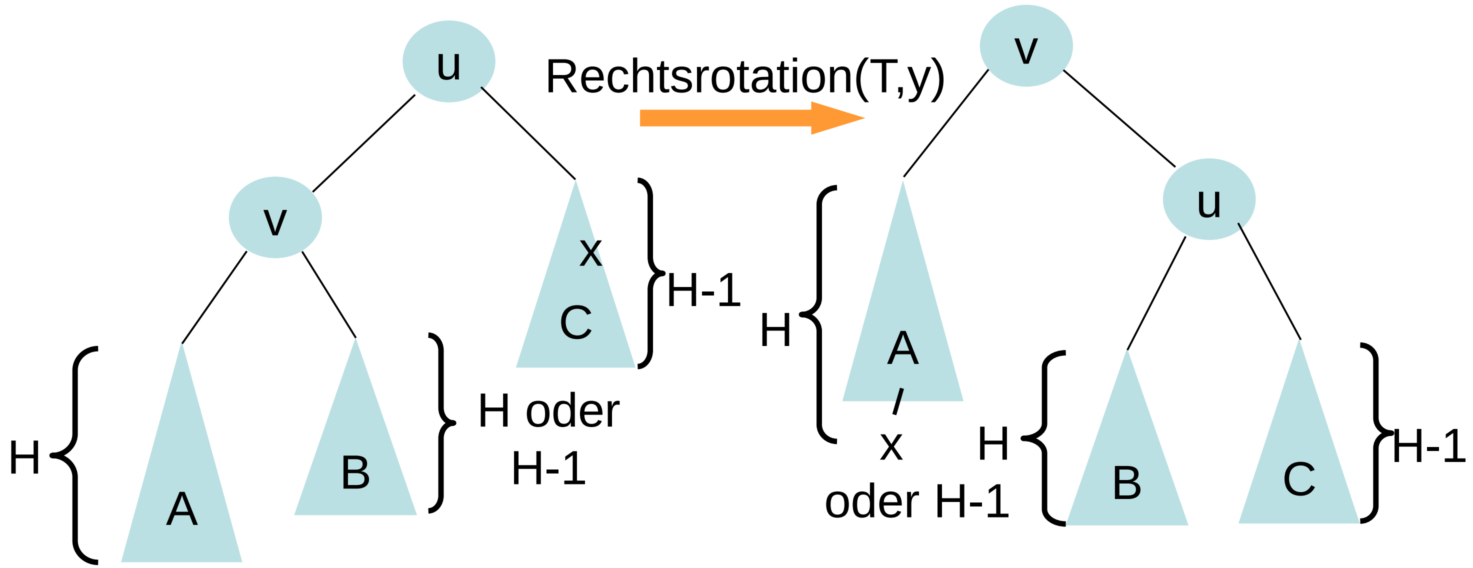
Balancierte binäre Suchbäume

Korrektheit:

Behauptung: Eine Rebalancierung bei u ist nur für h´(u)=h(u) möglich.

Beweis:

- Fall 1: einfache Rotation.
  Nach der Rotation ist daher h´(v)={h(u),h(u)-1}.

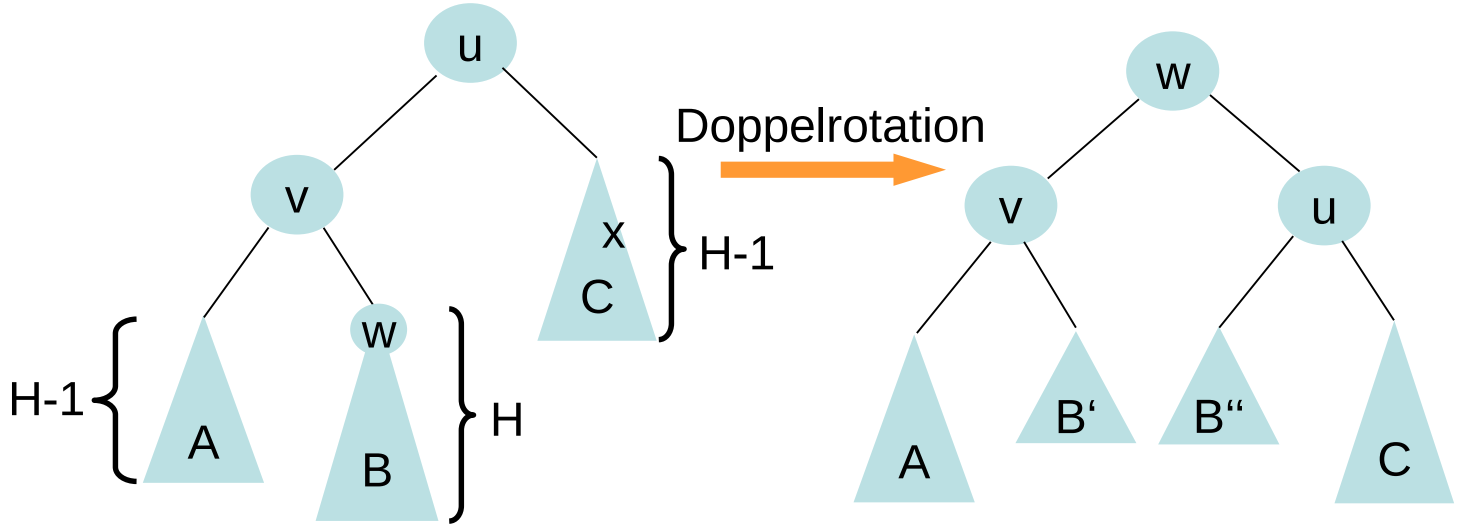


Korrektheit:

Behauptung: Eine Rebalancierung bei u ist nur für h´(u)=h(u) möglich.

Beweis:

- Fall 2: Doppelrotation.
  Kann auch nur passieren, wenn sich die Höhe von C verkleinert hat.
  In diesem Fall gilt dann aber, dass h´(u)=h(u).



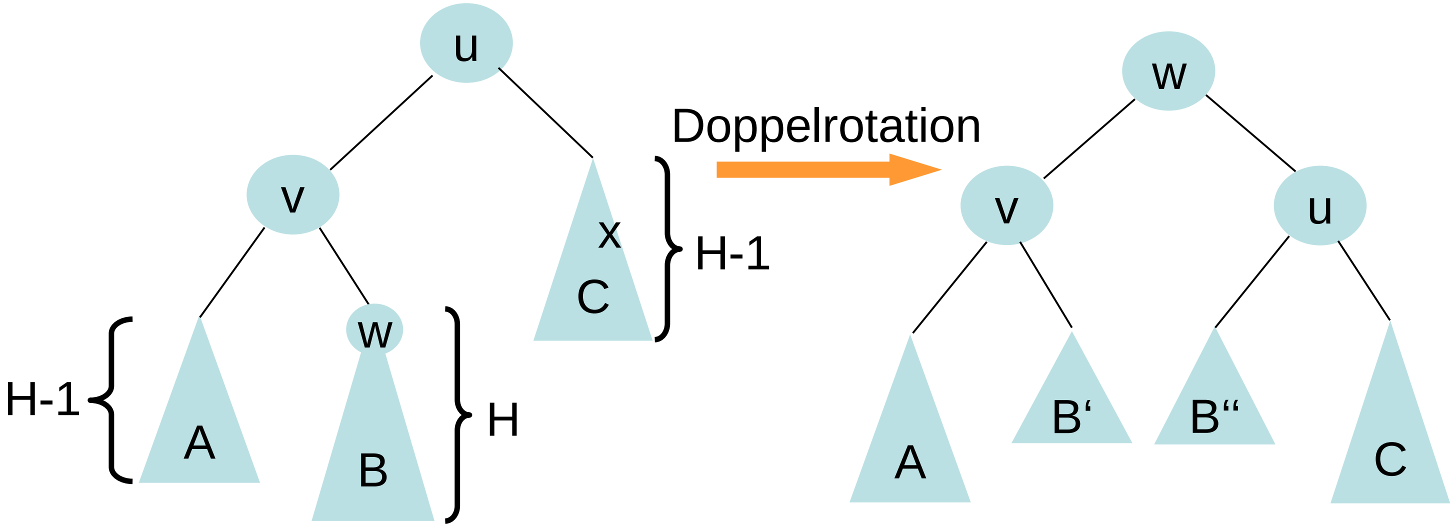
Balancierte binäre Suchbäume

Korrektheit:

Behauptung: Eine Rebalancierung bei u ist nur für h´(u)=h(u) möglich.

Beweis:

- Fall 2: Doppelrotation.
  Nach der Rotation ist daher h´(w)=h(u)-1.



Balancierte binäre Suchbäume

Korrektheit:

Behauptung: Eine Rebalancierung bei u ist nur für h´(u)=h(u) möglich.

Die Fälle im Beweis haben weiterhin gezeigt, dass

- Rotationen nur möglich sind, wenn für einen Knoten u gilt, dass h´(u)=h(u), während für ein Kind v von u, h´(v)=h(v)-1 ist.
- Weiterhin gilt nach den Rotationen, dass schlimmstenfalls h´(u)=h´(u)-1 gilt, so dass die Höhendifferenz bei den Vorfahren von u nicht schlimmer als 2 werden kann.
- Allerdings können im Gegensatz zum Insert beim Remove logarithmisch viele Rotationen notwendig sein. Da jede Rotation aber nur konstanten Aufwand hat, ist das nicht weiter schlimm.

Balancierte binäre Suchbäume

Satz 12.3

Mit Hilfe von AVL-Bäumen kann man Suche, Einfügen, Löschen, Minimum und Maximum in einer Menge von n Zahlen in $\Theta(\log n)$ Laufzeit durchführen.

Zusammenfassung und Ausblick:

- Effiziente Datenstruktur für das Datenbank Problem mit Hilfe von Suchbäumen
- Kann man eine bessere Datenstruktur finden?
- Was muss man ggf. anders machen?
  (untere Schranke für vergleichsbasierte Strukturen)

Changelog

23.05.16: fast alle Folien ab Folie 33

27.05.16: Folien 57-60 (neu), ab 83 neu oder überarbeitet