ADT Wörterbuch (Dictionary)

W: Kollektion von Schlüsseln, kein Schlüssel kommt mehrfach vor

search(x, W) ist x in W? (falls ja: liefere Ort im Speicher)

insert(x, W) füge x in W ein (melde, falls schon drin)

remove(x, W) entferne x aus W

Datenstruktur?

unsortiertes Array: alle Ops $O(n)$

sortiertes Array: search $O(\log n)$, insert / remove $O(n)$

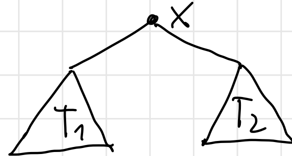
verkettete Liste: search / insert / remove $O(n)$ (egal ob sortiert oder nicht)

Heap: search $O(n)$

Ziel: alle Ops in $O(\log n)$ // n = # Schlüssel in W

Idee: Verwende Binärbaum

Schlüssel an der Wurzel soll uns sagen, ob wir im rechten oder linken Teilbaum suchen müssen



Suchbaumbedingung

Für jeden Knoten x:
- Alle Schlüssel in T_1 sind $< x$
- Alle Schlüssel in T_2 sind $> x$

Beispiel Suchbaum:

je nach Bedarf

parent

key = 7	
left	right

22, 7, 70, 3, 8, 27, 88, 12, 31, 89

null, null, null

Höhe h

Binärbaum: Jeder Knoten hat ≤ 2 Kinder

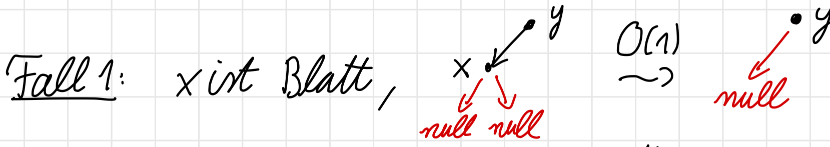
Search, Insert: $O(h)$

```
Search(x, p)   // Suche in Teilbaum unter p
    if p = null : Misserfolg
    else if p.Key = x : Erfolg
    else
        if x < p.Key : Search(x, p.left)
        else         : Search(x, p.right)
```

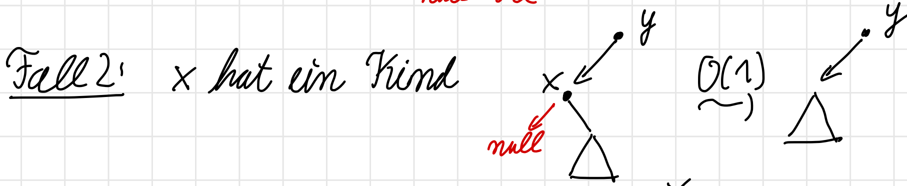
```
Insert(x): Search(x)
           Ersetze null-Blatt durch [x | null | null]   // falls nicht gefunden
```

remove(x) : erst search(x), dann Fallunterscheidung:

Fall 1: x ist Blatt,

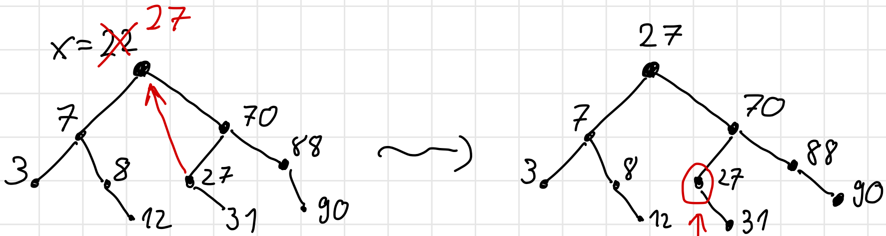


Fall 2: x hat ein Kind



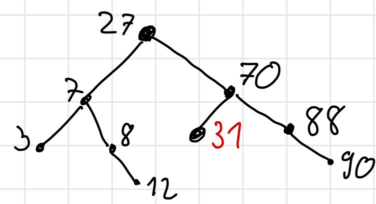
Fall 3: x hat zwei Kinder

Beispiel:
remove(22)



Womit können wir 22 ersetzen?
Mit nächstgrösserem Element in Baum:
gehe im rechten Teilbaum immer nach links, bis es kein linkes Kind mehr gibt. Laufzeit $O(h)$

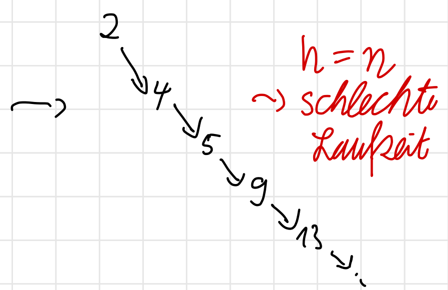
O(1) remove (einfach, da kein linkes Kind → Fall 1 oder 2)



→ Laufzeit von remove: $O(h)$

Problem: h kann gross werden

Beispiel: Baue Baum durch Einfügen einer sortierten Folge: 2, 4, 5, 9, 13, ...



Wie erhalten wir $h = O(\log n)$?

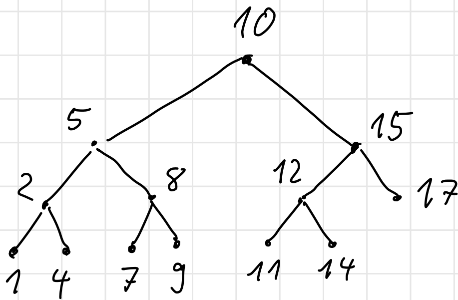
Verbleibendes Problem: Wie erhalten wir $h = O(\log n)$?

Idee 1: <u>vollständiger Binärbaum</u>

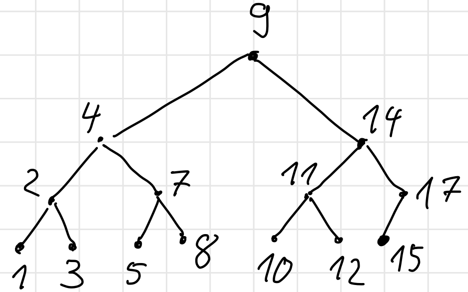
wie bei Heaps

→ zu starr, es gibt immer nur eine Möglichkeit

z.B. für 1, 2, 4, 5, 7, 8, 9, 10, 11, 12, 14, 15, 17

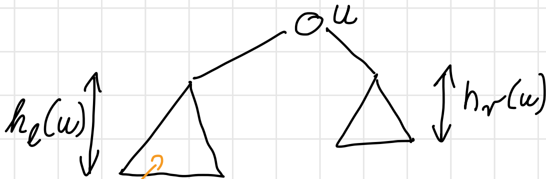


insert(3) ⇝



extrem viel Arbeit! $\Theta(n)$ worst case

Idee 2: mehr Flexibilität, folgende <u>AVL-Bedingung</u>:

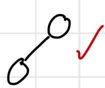


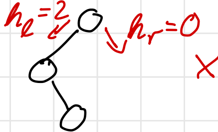
$AVL(u): |h_\ell(u) - h_r(u)| \leq 1$ für alle Knoten u

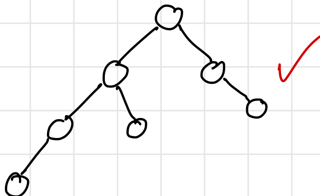
könnte leer sein, deshalb zählen wir hier Höhe(null) = 0, Höhe(∘) = 1, Höhe(∘–∘) = 2, ...

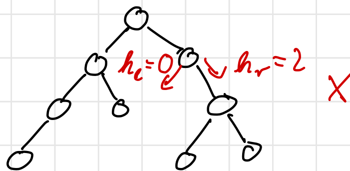
⇝ AVL-Bäume (Adelson-Velsky, Landis)

Beispiele:









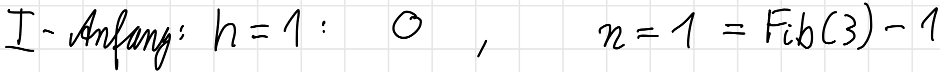
1) Wie ist die Höhe h? $O(\log n)$?

2) Wie erhält man die AVL-Bedingung?

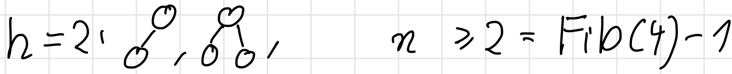
zu 1: <u>Satz</u>: Ein AVL-Baum der Höhe h hat $n \geq Fib(h+2)-1$ viele Knoten

<u>Beweis</u>: Induktion

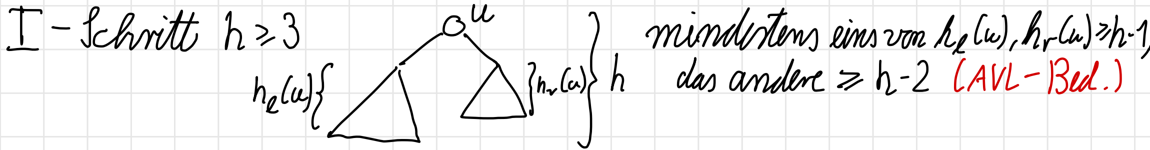
Fib: 1, 1, 2, 3, 5, ...



I-Anfang: $h=1$: o , $n = 1 = Fib(3)-1$



$h=2$: , $n \geq 2 = Fib(4)-1$



I-Schritt $h \geq 3$

mindestens eins von $h_l(u), h_r(u) \geq h-1$, das andere $\geq h-2$ (AVL-Bed.)

$\rightsquigarrow n \geq 1 + \#$ Knoten links $+ \#$ Knoten rechts

$\overset{I.-H.}{\geq} 1 + Fib(h-1+2) - 1 + Fib(h-2+2) - 1$

$= Fib(h+2) - 1$ ☒

Wir wissen $Fib(h+2) \geq \frac{1}{3} \cdot 1.5^{h+2}$ (Übung 2)

$\rightsquigarrow$ Ein AVL-Baum der Höhe h hat $n \geq Fib(h+2) - 1 \geq \frac{1}{3} \cdot 1.5^{h+2} - 1$ Knoten

$\rightsquigarrow 1.5^{h+2} \leq 3 \cdot (n+1)$

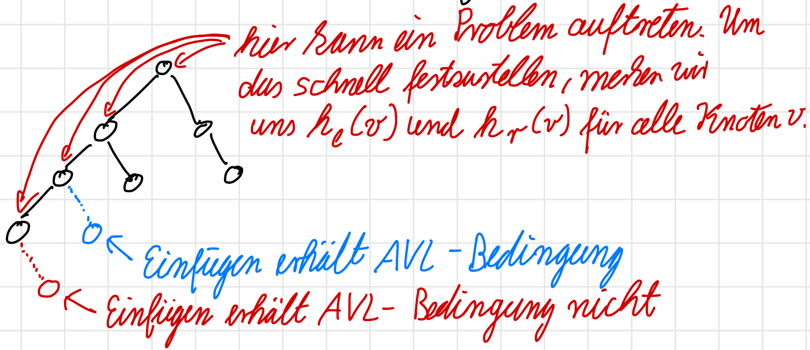
$\rightsquigarrow h+2 \leq \log_{1.5}(3 \cdot (n+1))$

$\rightsquigarrow h \leq O(\log n)$. $\rightsquigarrow$ search in $O(\log n)$

2) Rebalancieren: nach einem insert/remove könnte die AVL-Bedingung verletzt sein ⇝ reparieren

Wir schauen uns insert an. Remove geht ähnlich.

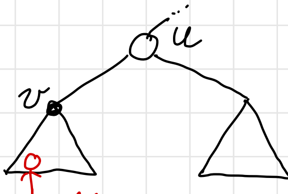
Beispiel:



Probleme können nur bei (direkten) Vorfahren des neuen Knotens auftreten.

- Durchlaufe die Vorfahren von unten nach oben
- Repariere AVL-Bedingung, wo nötig
- Aktualisiere gegebenenfalls h_ℓ bzw. h_r

Reparatur der AVL-Bedingung

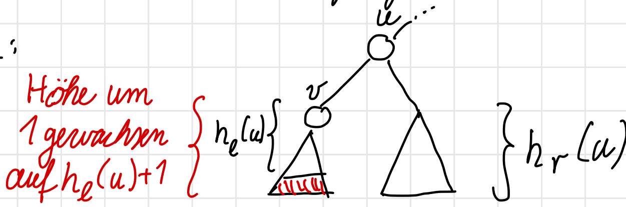


(falls v rechtes Kind von u: analog)

Fall 1: Höhe von v bleibt unverändert
→ kein Problem in u, keine Rekursion nötig

Reparatur der AVL-Bedingung, allgemeine Situation

Fall 2:



Fall 2A: $h_r(u) = h_\ell(u)+1$

→ Erhöhe $h_\ell(u)$ um 1. Höhe von u bleibt unverändert
→ keine weitere Rekursion nötig

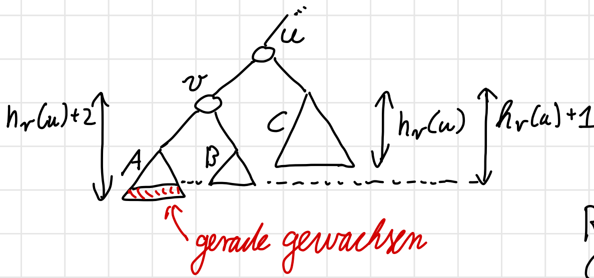
Fall 2B: $h_r(u) = h_\ell(u)$

→ Erhöhe $h_\ell(u)$ um 1. AVL-Bedingung an u erfüllt, aber Höhe von u erhöht sich
→ prüfe Elternknoten von u (Rekursion nötig)

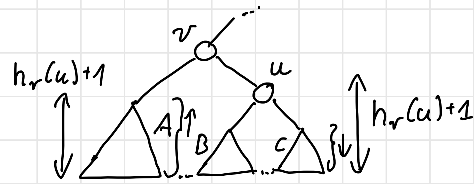
Fall 2C: $h_r(u) = h_\ell(u)-1$ → AVL-Bedingung an u nun verletzt

weitere Fallunterscheidung: Ist die Höhe von v wegen linkem oder wegen rechtem Teilbaum gewachsen?

Fall 2Ca: linker Teilbaum, $h_\ell(v) = h_r(v)+1$ (nach Aktualisierung)



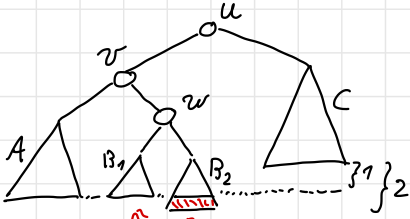
Rotation $O(1)$



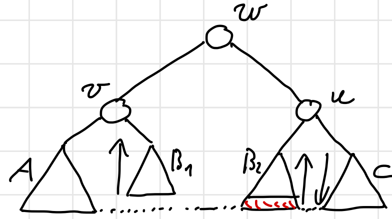
Gesamtbaum wieder so hoch wie vor insert:
neue Höhe von v = alte Höhe von u
→ keine weitere Rekursion nötig

Suchbaumbedingung ist weiterhin erfüllt.
AVL-Bedingung erfüllt.

Fall 2Cb: $h_\ell(v) = h_r(v) - 1$



Doppel-rotation $\xrightarrow{O(1)}$



entweder B_1 oder B_2 ist grade gewachsen (hier: B_2)

Suchbaumbedingung ist weiterhin erfüllt.

A bleibt auf selber Höhe
B_1, B_2 gehen 1 nach oben
C geht 1 nach unten
⇝ AVL-Bedingung erfüllt

Gesamtbaum ist wieder so hoch wie vor insert
⇝ keine weitere Rekursion nötig

Laufzeit von insert(x) ist $O(1)$ pro Level ⇝ $O(h) = O(\log n)$ insgesamt

Remove(x) geht ähnlich: balancieren mit Rotationen oder Doppel-rotationen
⇝ $O(\log n)$

GEHT ES BESSER?

JA, aber nur mit randomisierten Datenstrukturen (Hashing)
⇝ nächstes Semester

Aber: Suchbäume erlauben weitere Operationen, die mit Hashing nicht effizient gehen:
- Finde Minimum / Maximum
- Finde das i-kleinste Element (wenn wir uns zusätzlich für jeden Knoten die Knotenzahl im linken und rechten Teilbaum merken)

Suchbäume können auch für die ADT Liste eingesetzt werden ⇝ alle Ops in $O(\log n)$