Algorithmen & Datenstrukturen

Herbst 2018

Vorlesung 8

Vorlesung Gerüst:

	Algorithmen	Datenstrukturen
Entwurf	✓	heute
Analyse	✓	heute

① Datenstrukturen für abstrakte Datentypen (ADTs)

ADT: Objekte
Operationen

Beispiel: Studentendatensätze
Schlüssel = Matrikelnummer

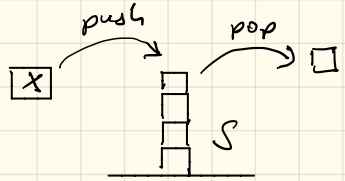
Beispiel: Objekte = Schlüssel $\in \mathbb{N}$

Datenstruktur =
Implementierung eines ADTs
Ziel: Effizienz

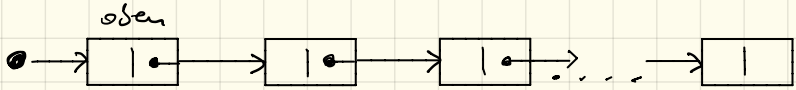
1. ADT Stapel (Stack)

push (x, S) legt x auf Stapel S
pop (S) entfernt (und liefert) oberstes Element
top (S) liefert oberstes Element
(isempty (S), emptystack → leerer Stapel)

Visualisierung:



Datenstruktur: verkettete Liste (linked list)



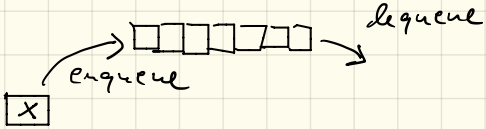
⇒ alle Ops in $O(1)$

Array geht auch, aber man muss max. Länge kennen

2. ADT Schlange (Queue)

enqueue (x, S)	füge x hinten an
dequeue (S)	entferne (und liefere) vorderstes Element

Visualisierung:



Datenstruktur: doubly linked list
+ Zeiger auf letztes Element
⇒ beide Ops in $O(1)$

3. ADT Priority Queue

insert (x, P)	füge x ein
extractmax (P)	lösche (und liefere) Maximum

Datenstruktur: Heap (Beispiel: heap sort)
⇒ beide Ops in $O(\log n)$

4. ADT Wörterbuch (Dictionary)

search (x, W)	ist x in W?
insert (x, W)	füge x in W ein (melde wenn schon drin)
remove (x, W)	entferne x von W

Datenstruktur?

Sortiertes Array | --- | 7 | 12 | 17 | 81 | --- |

Suche: $O(\log n)$, Einfügen/entfernen: $O(n)$

Unsortiertes Array Alle Ops $O(n)$

Linked List Alle Ops $O(n)$ (egal ob sortiert oder unsortiert)

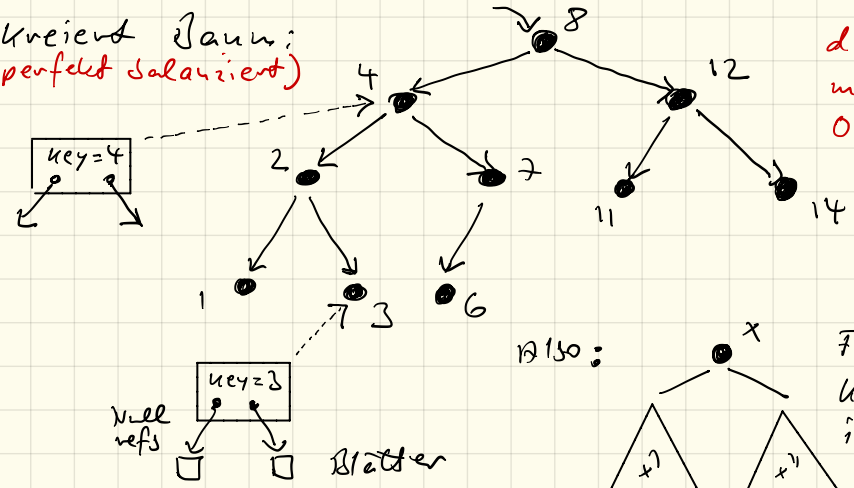
Heap Suche ist $O(n)$

Ziel: alle Ops in $O(\log n)$
Idee: verwende Baum

Suchbaum

Binäre Suche: | $x < x$ | x | $x > x$ |

Kreisend Baum:
(perfekt balanciert)



dann kann man in $O(\log n)$ suchen

Also:

Für alle Knoten x im Baum

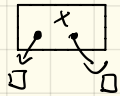
$x' < x < x''$

"Suchbaumbedingung"

```
Suche (x, p)
  if p = null : Misserfolg
  else if p.key = x : Erfolg
  else
    if x < p.key : Suche (x, p.left)
    else : Suche (x, p.right)
```

p: | key | |
|---|---|
| left | right |

Einfügen (x, p): Suche (x, p)

Erstes Blatt durch 

Suche Ops in $O(h)$, h = Höhe (Baum)

$\log_2(n) \leq h \leq n$ (n = Anzahl Schlüssel)

Problem: Baum kann sehr unbalanciert werden

Beispiel: Eingabe einer sortierten Folge

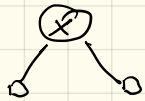
2, 4, 5, 9, 13, ...

ergibt 

Wie halten wir h klein? → später

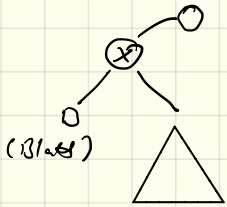
Entferne(x, p): zuerst Suche(x, p) in $O(h)$

Fall 1:



(Blatt) $\Rightarrow$ einfach löschen $O(1)$

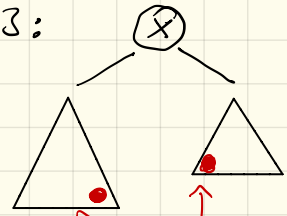
Fall 2:



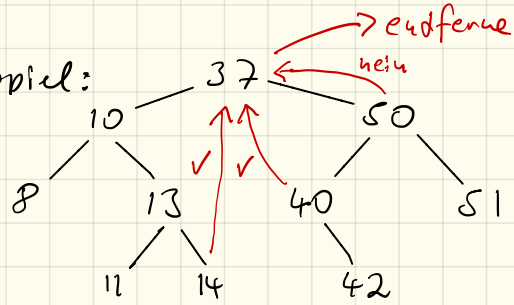
$\Rightarrow$

$O(1)$

Fall 3:



Beispiel:



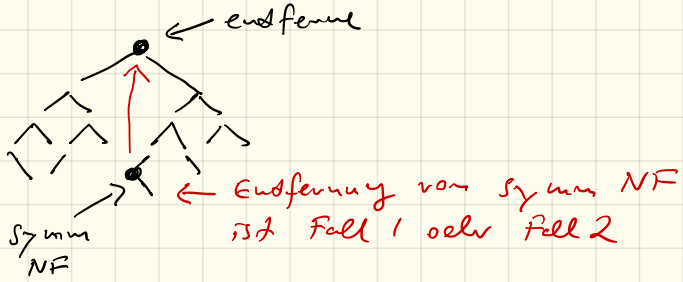
womit ersetzen wir 37 um die Suchbaumbedingung zu erhalten?

Symmetrischer Nachfolger: nächst grösseres Element im Baum

$\Rightarrow$ tausche x mit symm. NF

(symm NF: gehe von x einmal nach rechts und dann immer nach links bis zu einem Blatt) Blatt = Nullref

Visuell:

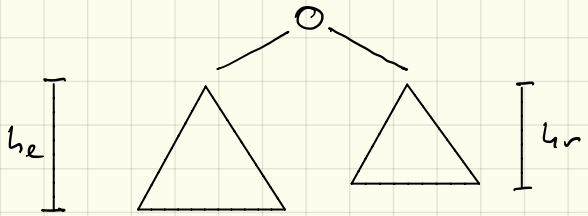


$\Rightarrow$ Endferne ist $O(h)$

Verbleibendes Problem: $h = \Theta(\log n)$ erhalten

Idee 1: erhalte perfekte Balancierung ist schwierig!

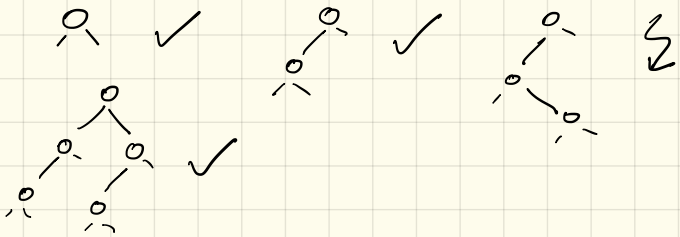
Idee 2: relaxiere und verlange nur die folgende Strukturbedingung



AVL-Bäume (Adelson-Velsky, Landis)

$|h_l - h_r| \leq 1$ für alle Knoten im Baum

Beispiele:



1.) Wie groß ist die Höhe h? (näher an $\log_2(n)$ oder n?)
2.) Wie erhält man die AVL Bedingung?

zu 1.) Idee: Bestimme eine untere Schranke für die Anzahl der Blätter und verwende:

Ein Binärbaum der n Schlüssel (innere Knoten) gespeichert hat $n+1$ Blätter. (Beweis: Induktion)

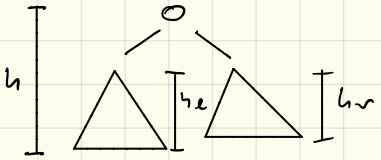
Also: $MB(h)$ = Minimale Blattzahl eines AVL Baums der Höhe h $\quad (\Rightarrow n \geq MB(h)-1)$

$MB(1) = 2$ = $FiS(3)$

$MB(2) = 3$ = $FiS(4)$

$MB(h) = MB(h-1) + MB(h-2)$

= $FiS(h+2)$



h_e, h_v: eine $= h-1$, andere $\geq h-2$

$\Rightarrow MB(h) = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^h\right)$, d.h. $n \geq \Omega\left(\left(\frac{1+\sqrt{5}}{2}\right)^h\right)$

genauer: $n \geq 1.6\ldots^h$ $\quad \approx 1/\log_2\left((1+\sqrt{5})/2\right)$

$\Rightarrow h \leq 1.44 \log_2(n)$ also höchstens 44% höher als perfekt balanzierte

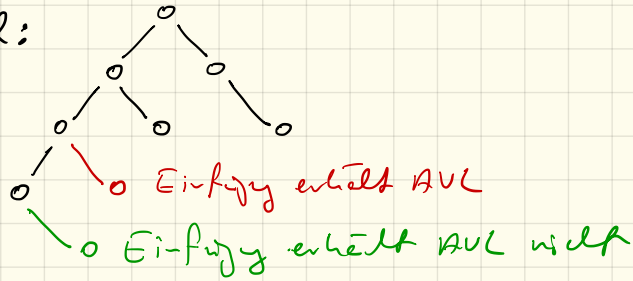
$\Rightarrow O(h) = O(\log n)$

Also: einfügen und dann:

1.) einfügen
2.) evtl. rebalancieren (AVL Bedingung wiederherstellen)

dazu genügt es alle Vorgänger des eingefügten Elements anzusehen ($O(h)$ viele)

Beispiel:

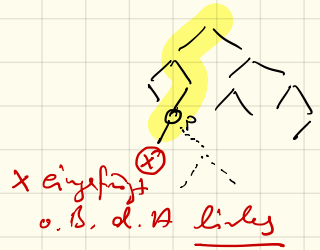


Wir merken uns in jedem Knoten p

$bal(p) = h_r - h_l$

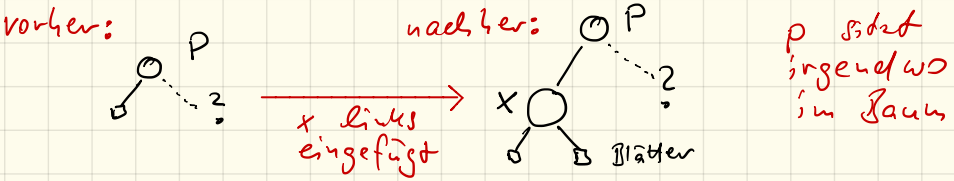
-1: linker Teilbaum höher
0: beide gleich hoch
1: rechter Teilbaum höher

Rebalancieren:



gels: muss schlimmstenfalls gesetzt werden + rebalanciert

Einfügen von x (o.B.d.A links, rechts ist analog):



Fälle: 1.) $bal(p) = -1$ nicht möglich

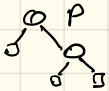
2.) $bal(p) = 0$ d.h. vorher 

$bal(x) = 0$

Höhe Teilbaum $bal(p) = -1$

p ist gewachsen! `upin(p)` (up after insertion)

nötig für:
- update von bal in Vorgängern
- evtl. Rebalancierung

3.) $bal(p) = +1$ d.h. vorher 

$bal(x) = 0$

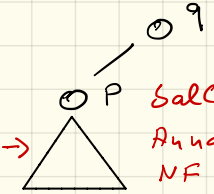
Höhe TB p nicht gewachsen $bal(p) = 0$ fertig

In allen 3 Fällen ist TB p AVL!

Der Aufruf `upin(p)` gilt Invariante
- $bal(p) \neq 0$
- Höhe TB p ist gewachsen
- p hat Vorgänger (sonst ist Aufruf unnötig)

Beschreibung upin(p)

Situation:



Höhe gewachsen +1

bal(p) ≠ 0

Annahme: p ist linker NF von q. Rechter NF geht analog.

Fälle: 1.) bal(q) = +1

bal(q) = 0 TB q ist AVL

fertig

2.) bal(q) = 0

Höhe TB q gewachsen

bal(q) = -1 TB q ist AVL

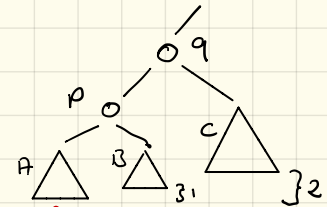
upin(q)

3.) bal(q) = -1 TB q ist nicht mehr AVL

→ Ansatz nötig

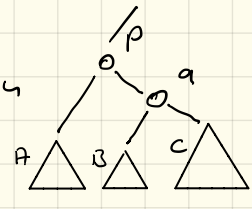
bal(p) = -1 oder +1

3a.) bal(p) = -1



* wurde hier eingefügt

Rotation O(1)



bal(p) = 0

bal(q) = 0

fertig

TB p hat gleiche Höhe wie TB q vor einfügen

⇒ upin nicht nötig

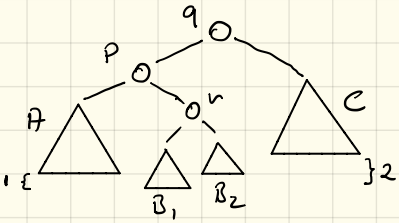
A: ging 1 nach oben

B: gleich

C: 1 nach unten

Suchbaumbedingung erfüllt!

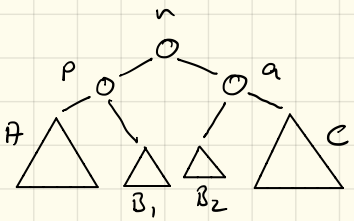
3S.) $bal(p) = +1$



x wurde in B_1 oder B_2 eingefügt (hier: B_1)

Doppel-notation

$O(1)$



$bal(r) = 0$

$bal(q) = 1$

$bal(p) = 0$

fertig

TB r hat gleiche Höhe wie TB q vor einfügen

⇒ upin nicht nötig

A bleibt in Höhe

B_1 geht 1 nach oben

B_2 geht 1 nach oben

C geht 1 nach unten

Suchbaumbedingung erfüllt!

also: Einfügen ist $O(\log n)$

Entfernen: geht ähnlich in $O(\log n)$