

datastructuren en algoritmen

2011 09 19

college 5

totnutoe

- **lineaire datastructuren**
stacks, queues, vectoren, lijsten, rijtjes, priority queues
- **hierarchische datastructuren**
bomen, binaire bomen, heaps
- **algoritmen**
sorteren, heap-sort, selection sort, insertion sort
- **hashing**

schema

- recap
- geordend dictionary
- binary search
- binaire zoekbomen
- AVL bomen
- materiaal

schema

- recap
- geordend dictionary
- binary search
- binaire zoekbomen
- AVL bomen
- materiaal

geordend dictionary

- collectie van key-element items
- we zoeken met de key
- operaties: zoeken, toevoegen, verwijderen
- nu: de keys zijn (totaal) geordend

geordend dictionary: ADT

extra operaties bij het ADT voor ongeordend dictionary:

- `closestKeyBefore(k)`
retourneert de key van het item met de grootste key $\leq k$
- `closestElementBefore(k)`
retourneert het element van het item met de grootste key $\leq k$
- `closestKeyAfter(k)`
retourneert de key van het item met de kleinste key $\geq k$
- `closestElementAfter(k)`
retourneert het element van het item met de kleinste key $\geq k$

ongeordende en geordende dictionaries

- geen ordening op de keys: hashing of log file
- wel ordening op de keys: nu

schema

- recap
- geordend dictionary
- **binary search**
- binaire zoekbomen
- AVL bomen
- materiaal

binary search: voorbeeld

- we zoeken een naam k in een telefoonboek met 25000 namen
- zoek het midden m
key(m) = k dan: klaar
key(m) < k dan: zoek in rechterhelft
key(m) > k dan: zoek in linkerhelft
- zonder ordening: orde 25000 stappen
met ordening: orde 15 stappen
door de ordening kunnen we makkelijker zoeken

binary search: pseudocode

input: array A storing n integers in ascending order

output: **true** if A contains x and **false**, otherwise

Algorithm binSearch(A, n, x):

$low := 0$

$high := n - 1$

while $low \leq high$ **do**

$mid := \lfloor low + high/2 \rfloor$

$y := A[mid]$

if $x < y$ **then** $high := mid - 1$

if $x = y$ **then return true**

if $x > y$ **then** $low := mid + 1$

return false

binary search

idee: hak bij elke stap de zoekruimte in tweeën
en zoek verder in de goede helft

complexiteit van binary search via recurrente betrekking:

$$T(n) = \begin{cases} 1 & \text{als } n = 1 \\ T(\frac{n}{2}) + 1 & \text{als } n > 1 \end{cases}$$

lookup table

geordende dictionary geïmplementeerd met gesorteerd array

- in $\mathcal{O}(n)$: insertItem(k , e), removeElement(k)
(in het ergste geval moeten $n/2$ items opgeschoven worden)
dus updates zijn duur
- in $\mathcal{O}(\log n)$: FindElement(k)

vergelijk log file (ongeordend) en lookup table (geordend)

	log file	lookup table
zoeken	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
toevoegen	$\mathcal{O}(1)$	$\mathcal{O}(n)$
verwijderen	$\mathcal{O}(n)$	$\mathcal{O}(n)$

lookup table: wanneer gebruiken?

lookup table voornamelijk handig bij

- kleine database
- zelden toevoegen of verwijderen

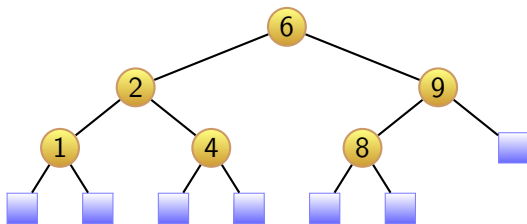
andere gevallen: de binaire zoekboom

schema

- recap
- geordend dictionary
- binary search
- **binaire zoekbomen**
- AVL bomen
- materiaal

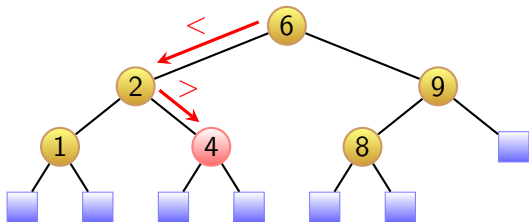
binaire zoekboom: eigenschappen

- op de interne knopen staan items, dus paren key-element (meestal schrijven we alleen de keys)
- de keys zijn geordend
- voor elke knoop n met key k geldt:
zijn linker subboom bevat alleen keys kleiner-gelijk k
zijn rechter subboom bevat alleen keys groter-gelijk k
- dus: met inorder traversal vinden we een niet-dalende rij



zoeken in binaire zoekboom: voorbeeld

zoek een knoop met key 4



zoeken in binaire zoekboom: idee algoritme

we zijn in knoop v en we zoeken key k

TreeSearch(k, v)

- v external dan return v
- v internal dan drie gevallen:
 - $k = \text{key}(v)$ dan return v
 - $k < \text{key}(v)$ dan return TreeSearch($k, \text{leftChild}(v)$)
 - $k > \text{key}(v)$ dan return TreeSearch($k, \text{rightChild}(v)$)

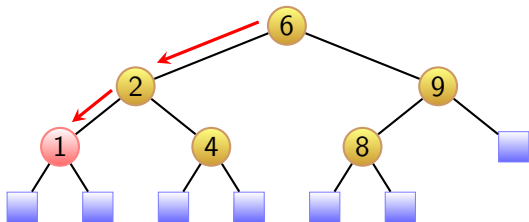
zoeken in binaire zoekboom: vervolg

FindElement(k)

- $v := \text{TreeSearch}(k, T.\text{root}())$
- v external dan: return NoSuchKey
 v internal dan: return element(v)

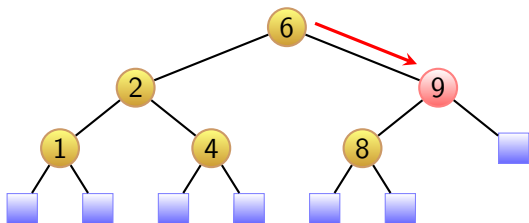
binaire zoekboom: zoek kleinste key

loop zover mogelijk naar links



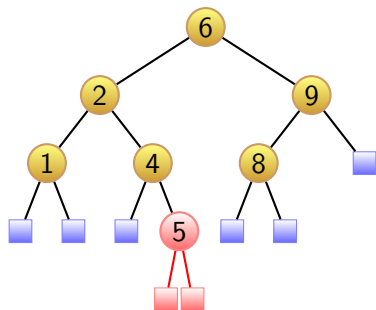
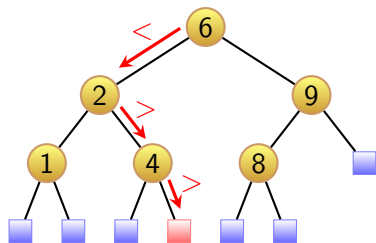
binaire zoekboom: zoek grootste key

loop zover mogelijk naar rechts



binaire zoekboom toevoegen: voorbeeld

voeg knoop met key 5 toe

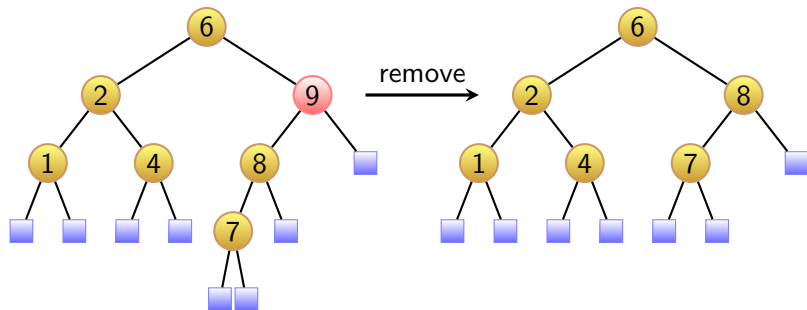


binaire zoekboom: toevoegen

insertItem(k, e)

- zoek de juiste plek:
 $w := \text{TreeSearch}(k, T.\text{root}())$
- w external dan: expandeer tot interne knoop met (k, e)
 w internal dan: $w := \text{TreeSearch}(k, \text{leftChild}(w))$

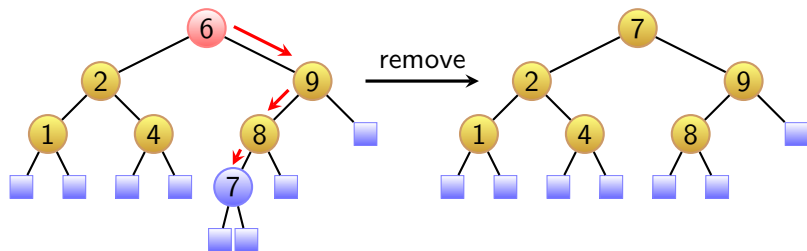
binaire zoekboom: voorbeeld verwijder boven external



binaire zoekboom: verwijder boven external

- linker opvolger is external dan:
vervang knoop door rechter subboom
- rechter opvolger is external dan:
vervang knoop door linker subboom

binaire zoekboom: voorbeeld verwijderen



binaire zoekboom: verwijderen

zoek eerst de knoop n

als niet in de boom dan klaar

als wel in de boom dan:

- als boven external dan: doe `removeAboveExternal(n)`
- als niet boven external dan:
zoek de volgende knoop voor inorder (kleinste in rechter subboom) m
vervang de key van n door de key van m
verwijder m met `removeAboveExternal`

binaire zoekboom: tijdscomplexiteit

(BST = binary search tree = binaire zoekboom)

	log file	lookup table	BST
zoeken	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(h)$
toevoegen	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(h)$
verwijderen	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(h)$

hoe groot is h ?

worst-case: $\mathcal{O}(n)$; best-case: $\mathcal{O}(\log n)$

dus: beperk de hoogte! maar hoe?

binaire zoekbomen

- zoeken
- toevoegen
let op als de key er al in staat
- verwijderen
let op als de te verwijderen knoop niet boven een external zit
- alledrie de operaties in $\mathcal{O}(\text{hoogte})$
probeer dus de hoogte te beperken met gebalanceerde bomen
- puzzel: hoeveel binaire zoekbomen zijn er met $1, \dots, n$?

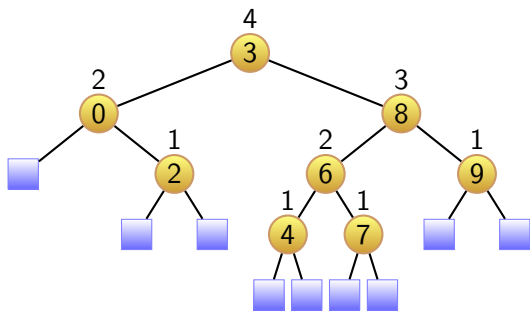
schema

- recap
- geordend dictionary
- binary search
- binaire zoekbomen
- **AVL bomen**
- materiaal

AVL boom: eigenschappen

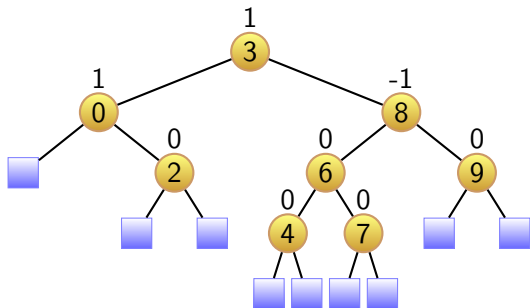
binaire zoekboom die gebalanceerd is
voor elke interne knoop geldt:

verschil tussen hoogte links en hoogte rechts is ten hoogste 1



balance factor

- **balance factor:** hoogte rechter min hoogte linker
- **AVL boom** als voor elke knoop: balance factor is -1 of 0 of 1



hoogte van een AVL boom

hoogte van een AVL boom met n interne knopen is in $\mathcal{O}(\log n)$

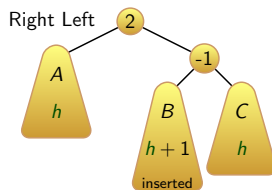
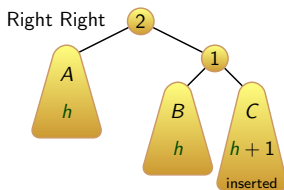
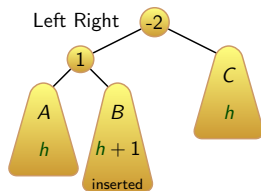
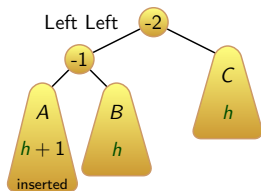
bewijs: zie boek, niet voor tentamen

AVL bomen: zoeken

- precies als in binaire zoekboom
- in $\mathcal{O}(\text{hoogte})$ dus in $\mathcal{O}(\log n)$

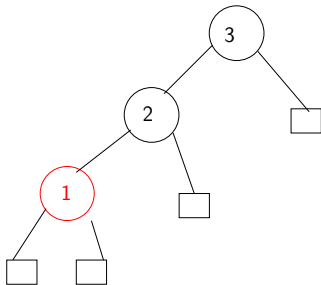
AVL bomen: toevoegen

- stap 1: zoals bij binaire zoekbomen
- stap 2: herbalanceer zo nodig
- de vier gevallen (waarom??) van ongebalanceerde bomen na toevoegen:

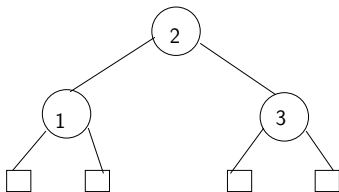


voorbeeld: links-links

toevoegen levert links-links ongebalanceerde boom:

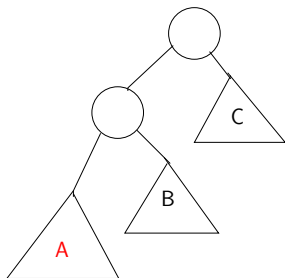


herbalanceren met single rotation:

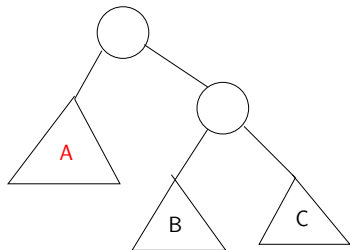


links-links algemeen

links-links ongebalanceerde boom



herbalanceren met single rotation: (vergelijk $(ab)c = a(bc)$)



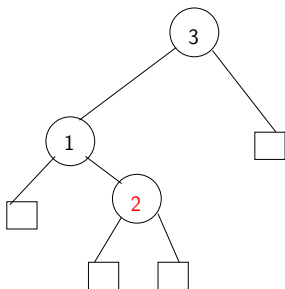
symmetrie: rechts-rechts

rechts-rechts ongebalanceerde boom

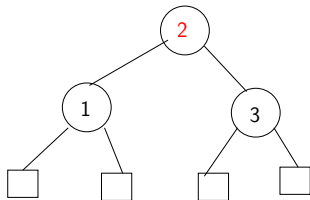
herbalanceren met single rotation

voorbeeld: links-rechts

toevoegen levert links-rechts ongebalanceerde boom:

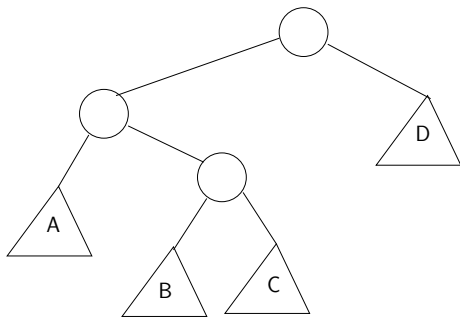


herbalanceren met double rotation:

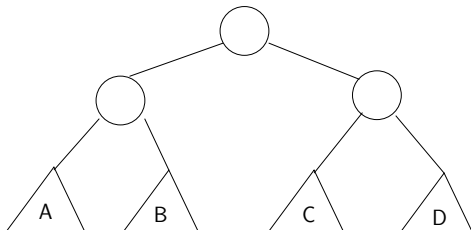


links-rechts algemeen

links-rechts ongebalanceerde boom



herbalanceren met double rotation



symmetrie: rechts-links

rechts-links ongebalanceerde boom

herbalanceren met double rotation

herbalanceren na toevoegen

- loop van de nieuwe knoop naar de wortel
- herbalanceer de eerste knoop met balance factor 2 of -2
- er zijn 4 gevallen (let op symmetrie)
- voor elk geval is een herbalanceer-regel

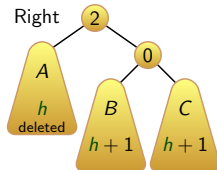
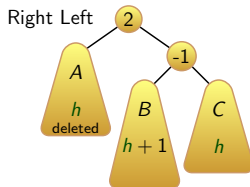
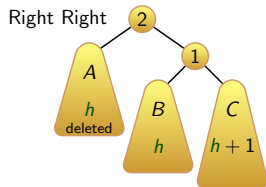
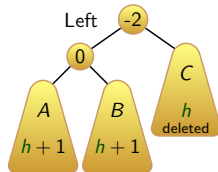
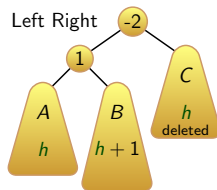
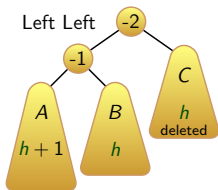
toevoegen: tijdscomplexiteit

er is ten hoogste 1 herbalanceer-stap nodig (waarom??)

toevoegen is in $\mathcal{O}(\log n)$

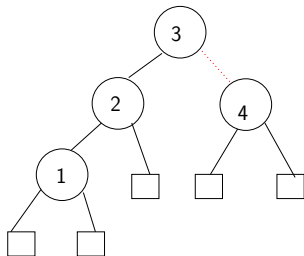
AVL bomen: verwijderen

- stap 1: zoals bij binaire zoekbomen
- stap 2: herbalanceer zo nodig
- de zes gevallen van ongebalanceerde bomen na toevoegen:

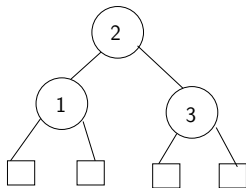


voorbeeld: links

verwijderen levert ongebalanceerde links



herbalanceren met single rotation



symmetrie: rechts

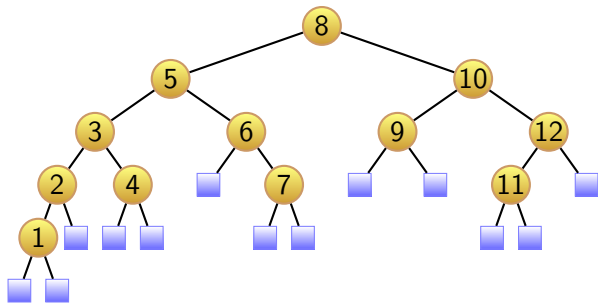
symmetrie geeft ongebalanceerde rechts boom

herbalanceren met single rotation

AVL bomen: verwijderen

- meer gevallen van ongebalanceerde bomen na update
- niet meer herbalanceer-regels nodig
- na verwijderen mogelijk meer dan een herbalanceer-stap nodig!

opgave: verwijder knoop met key 9



AVL boom: verwijderen

stap 1: zoals bij binaire zoekboom

stap 2: mogelijk moeten we herbalanceren

herbalanceren na verwijderen

- loop van de verwijder-plek naar de wortel
- herbalanceer de eerste knoop met balance factor 2 of -2
- er zijn 6 gevallen (waarvan twee nieuw, weer symmetrie)
- voor elk geval is een herbalanceer-regel

herbalanceren: vragen

- zijn soms meerdere regels toepasbaar?
- maakt het uit voor het eindresultaat welke regel je toepast?
- vinden we altijd een eindresultaat of zijn er loops?

AVL bomen: efficiency

- een rotatie is in $\mathcal{O}(1)$
- zoeken is in $\mathcal{O}(\log n)$
- toevoegen is in $\mathcal{O}(\log n)$
(zoeken, dan herbalanceren)
- verwijderen is in $\mathcal{O}(\log n)$ (NB)

overzicht

voor dictionaries:

	search	insert	remove
Log File	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Lookup Table	$\mathcal{O}(\log_2 n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
AVL Tree	$\mathcal{O}(\log_2 n)$	$\mathcal{O}(\log_2 n)$	$\mathcal{O}(\log_2 n)$

voor **geordende** dictionaries:

	closestAfter	closestBefore
Log File	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Lookup Table	$\mathcal{O}(\log_2 n)$	$\mathcal{O}(\log_2 n)$
AVL Tree	$\mathcal{O}(\log_2 n)$	$\mathcal{O}(\log_2 n)$

schema

- recap
- geordend dictionary
- binary search
- binaire zoekbomen
- AVL bomen
- **materiaal**

materiaal

- boek 3.1, 3.2

extra materiaal

- [wiki binaire zoekboom](#)
- [wiki AVL boom](#)