Föreläsning
Datastrukturer (DAT036)

Nils Anders Danielsson

2013-11-27

Idag

- Balanserade sökträd.
- Splayträd.
- Skipplistor.

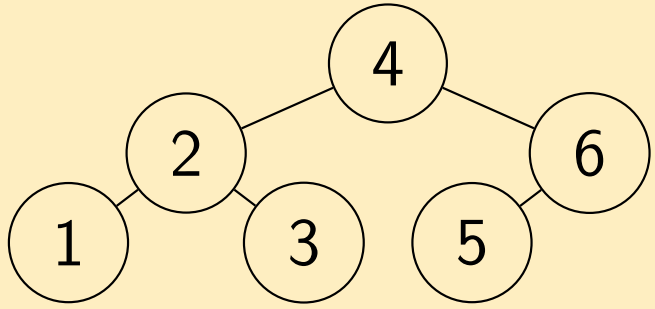
AVL-träd

AVL-träd

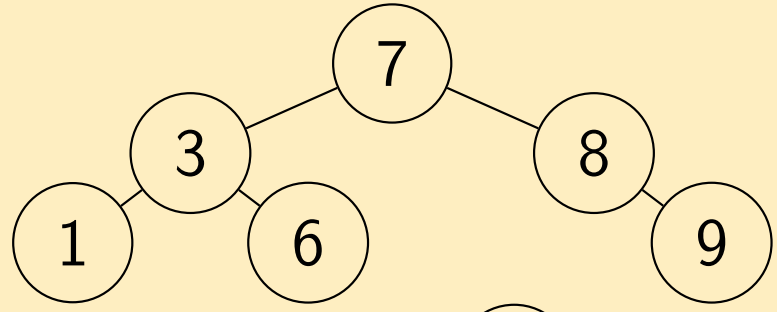
- Sökträd.
- Invariant (för varje nod):
  Vänster och höger delträd har samma höjd, ± 1.
- Höjd: $\Theta(\log n)$.
- Operationer som ändrar trädets struktur
  använder (ibland) *rotationer* för att
  återställa invarianten.

Vilka träd är AVL-träd?

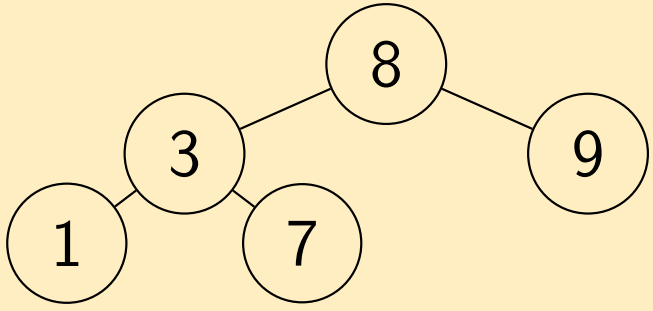
A:



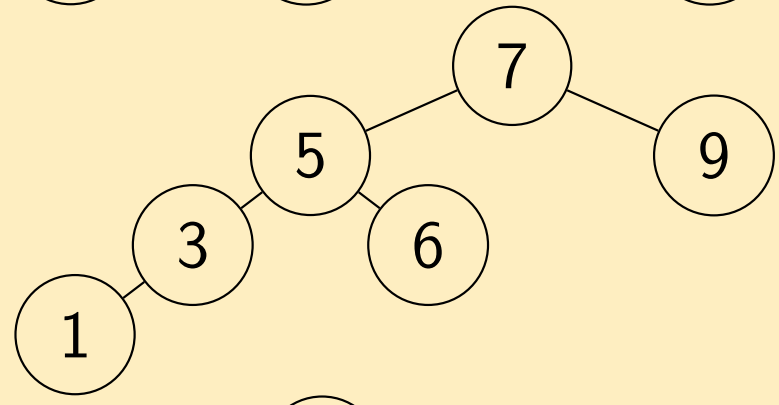
B:



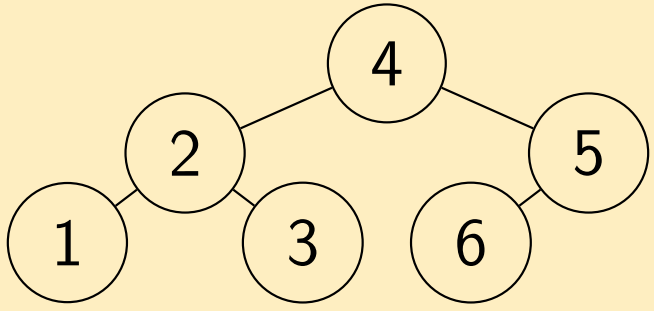
C:



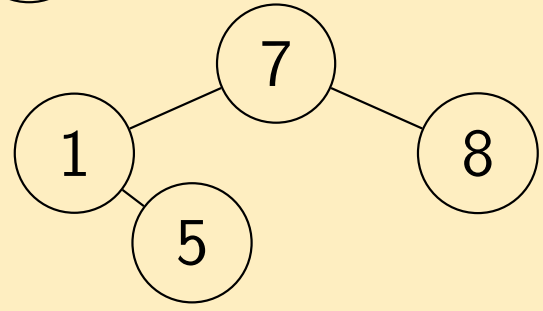
D:



E:



F:



AVL-träd

Implementation:

- Spara höjd i varje nod.
- Alternativ: Spara höjdskillnad (-1, 0 eller 1).
- Ibland används föräldrapekare.

`member`

Precis som för obalanserade binära sökträd.

insert

Skiss av en algoritm:

- Sätt in noden som vanligt.
- Gå tillbaka mot roten, uppdatera höjder.
- Vid första obalanserade noden: rotation.
- Antingen enkel- eller dubbelrotation (se tavlan eller boken).
- Det räcker med en rotation för att göra trädet balanserat.

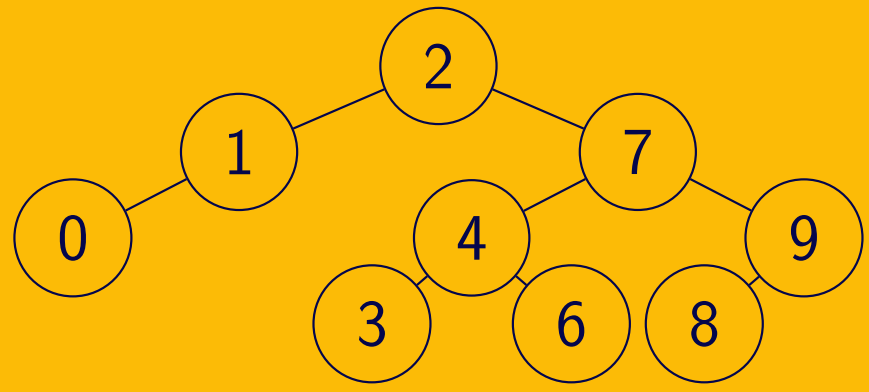
`delete`

- Kan utgå från algoritmen för obalanserade binära sökträd.
- Kan behöva rotera flera gånger.
- Alternativ: Lat borttagning.

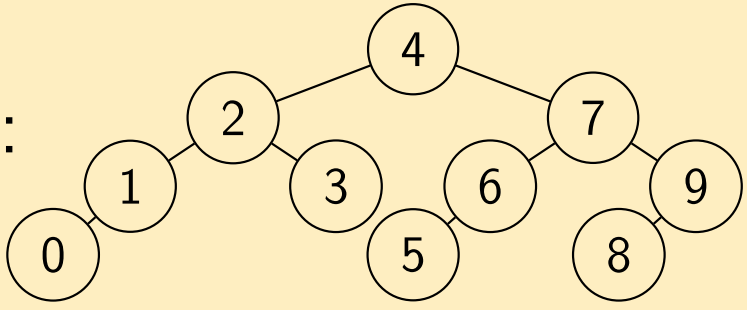
AVL-träd

Animerat: `http://www.qmatica.com/DataStructures/Trees/AVL/AVLTree.html`.

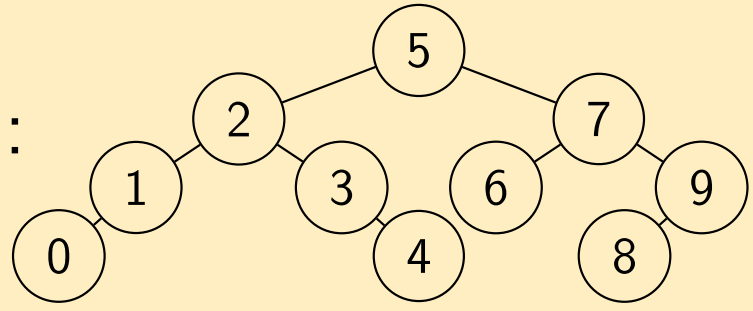
Vad blir resultatet av att sätta in 5 i följande AVL-träd?



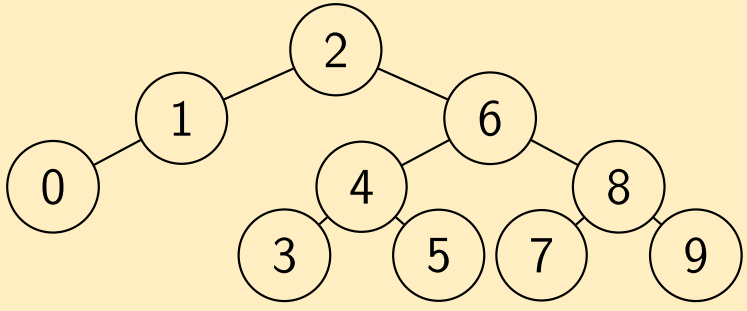
A:



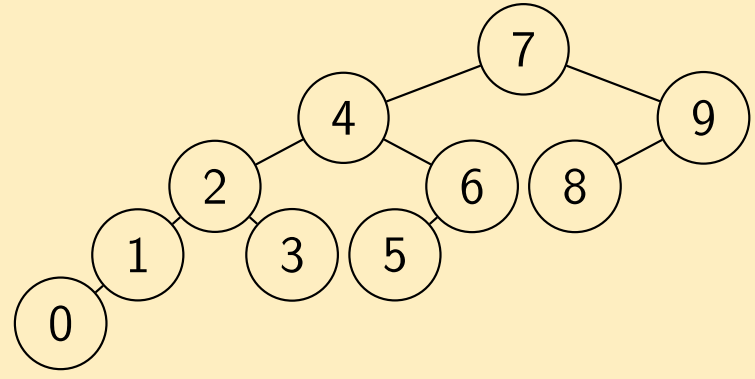
B:



C:



D:



Röd-svarta träd

Röd-svarta träd

- JDK 7: `TreeSet`, `TreeMap`.
- Höjd: $\Theta(\log n)$.

Röd-svarta träd

Invariant:

- Alla noder är svarta eller röda.
- Roten är svart.
- Röda noder har svarta barn.
- Alla (enkla) vägar från roten till noder med max ett barn innehåller lika många svarta noder.

Splayträd

Splayträd

- Inte nödvändigtvis balanserade.
- Träden omformas dynamiskt,
  även vid `member`/`lookup`.
- När man är långt ned i trädet passar man på att ”hämta upp” djupa noder.

Zig, zag

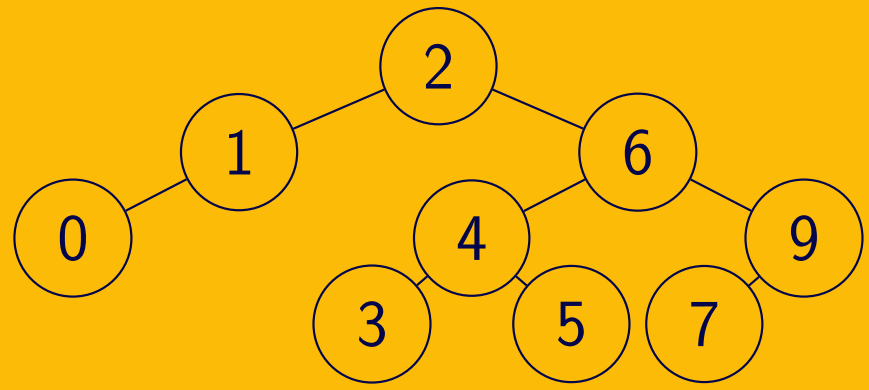
- En nod är i *zig*-position om den är ett vänsterbarn.
- *Zag*: Högerbarn.
- *Zig-zag*: Högerbarn till vänsterbarn.
- Och så vidare.

Splayträd

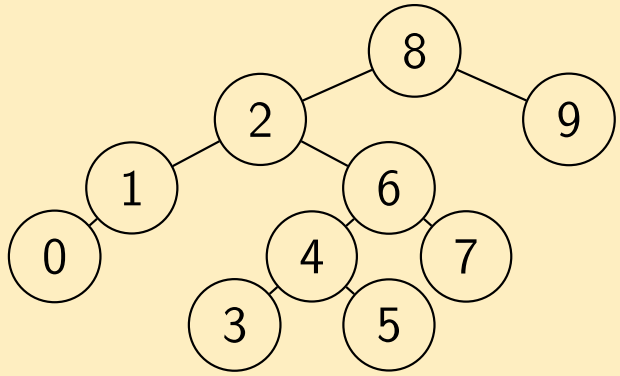
”Hämta upp” (splaya) genom att rotera
längs med sökvägen:

- Zig, zag (för barn till roten): Enkelrotationer.
- Zig-zag, zag-zig: Dubbelrotationer.
  Rotera upp noden, rotera upp noden igen.
- Zig-zig, zag-zag:
  En annan sorts dubbelrotationer.
  Rotera först upp föräldern, sedan noden.

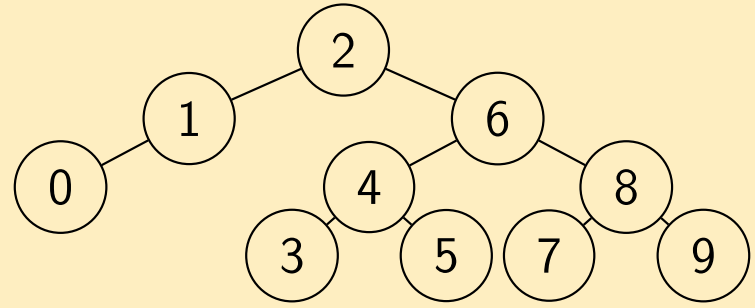
Vad blir resultatet av att sätta in 8 i följande splayträd?



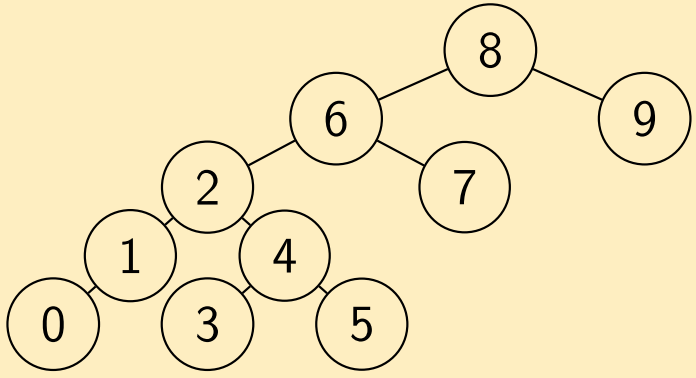
A:



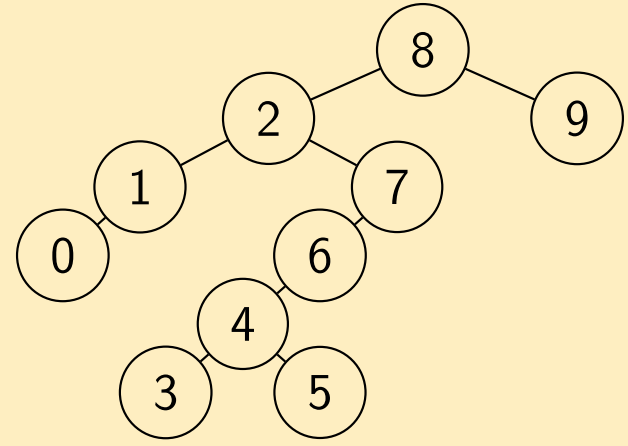
B:



C:



D:



Splayträd

Borttagning (en variant):

- Tomt träd: Klar.
- Leta upp noden, och splaya upp den till roten (eller splaya upp sista noden som besöktes och avbryt).
- Om högra delträdet tomt: Ta bort roten.
- Annars: Splaya upp högra delträdets minsta element till högra delträdets rot, byt ut roten mot högra delträdets rot.

Splayträd

Tidskomplexitetsanalys:

- Lite komplicerad.
- Splaya upp en nod: $O(\log n)$ amorterat.
- `find-min`, `delete-min`: $O(\log n)$ amorterat.
- `member`, `insert`, `delete`: $O(\log n)$ amorterat (om jämförelser tar konstant tid).
- Antagande: Enkeltrådad användning.

B-träd

B-träd

- Vad händer om en avbildning inte får plats i minnet, och lagras på en hårddisk e d?
- Läsa från hårddisk: Kanske $\sim 10^6$ gånger långsammare än CPU-instruktion.
- Vår modell fungerar inte så bra.
- Alternativ modell: Räkna diskoperationer, CPU-instruktioner gratis.
- OK att göra något (rimligt) komplicerat om vi kan minska antalet diskoperationer.

B-träd

Några idéer:

- M-ära träd istället för binära:
  - Kompletta träd grundare ($\log_M n$).
  - $\leq M - 1$ nycklar per intern nod.
- Gör noder som ska sparas på disk så stora att de fyller ett diskblock (när de är fulla).

B-träd

- Används i filsystem och databaser.

Skipplistor

Skipplistor

- Sorterad länkad lista: långsam sökning.
- Kanske kan lägga till fler länkar.
- Om nod $i2^k$ har länk till nod $(i+1)2^k$:
  snabb sökning, långsam insättning.
  - Höjd (maximalt antal länkar från en nod):
    $\Theta(\log n)$.
  - `member`: $O(\log n)$
    (om jämförelser tar konstant tid).
  - `insert`: $O(n)$.

Skipplistor

Skipplistor:

- Samma idé, men slumpmässig struktur (samt vaktpost av maximal höjd i början).
- Insättning: Slumpmässig höjd på noden.
- Stanna på höjd 1 med sannolikhet $1 - p$.
- Stanna på höjd 2 med sannolikhet $1 - p$.
- ...

Skipplistor

- Sannolikhet för höjd h: $P(h) = (1-p)p^{h-1}$.
- Förväntad höjd: $\sum_{h=1}^{\infty} hP(h) = 1/(1-p)$.
- Rekommenderade val av p: $1/4$, $1/2$.

Vad är det förväntade antalet noder med höjd $\geq h$ (exklusive vaktposten)?

- pn^{h-1}.
- $n(1-p)p^{h-1}$.
- np^{h-1}.
- n.
- $n(1-p)^{h-1}$.

Skipplistor

- *Förväntad* tidskomplexitet för `insert`, `member`, `delete`, för konstant p: $O(\log n)$ (om jämförelser tar konstant tid).
- I praktiken: Maxhöjd $\log_{1/p} N$, där N är listans största tänkbara storlek.

Sammanfattning

Idag:

- Balanserade sökträd.
- Splayträd.
- Skipplistor.

Nästa gång: Mer om sortering.