Föreläsning 9
Datastrukturer (DAT037)

Fredrik Lindblad[1]

27 november 2017

[1]Slides skapade av Nils Anders Danielsson har använts som utgångspunkt. Se
`http://www.cse.chalmers.se/edu/year/2015/course/DAT037`

Innehåll

- Obalanserade sökträd
- Balanserade sökträd
  - AVL-träd
  - Röd-Svarta träd
  - B-träd

Sökträd

Binära sökträd

Binära träd med sökträdsegenskapen:

- Tomma binära träd är sökträd.
- Ett icketomt binärt träd är ett sökträd om:
  Vänster och höger delträd är sökträd och
  alla element i vänstra delträdet $<$
  elementet i roten $<$
  alla element i högra delträdet.
- Jämför heapsorteringsegenskapen som bara behöver kontrolleras lokalt.

Binära sökträd

- Behöver kunna jämföra element,
  t ex med komparator.
- Komparatorn antas implementera en *strikt total ordning*:
  - Om $x < y$ och $y < z$ så är $x < z$.
  - Exakt en av följande gäller:
    $x < y$, $x = y$, $x > y$.

Binära sökträd

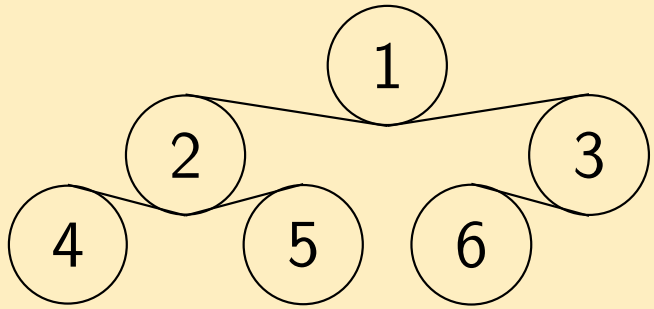
Sökträd kan användas för att implementera utökad mängd-/avbildnings-ADT:

- Konstruerare: Tom mängd/avbildning.
- `member(k)`/`lookup(k)`.
- `insert(k)`/`insert(k,v)`.
- `delete(k)`.
- `find-min()`/`find-max()`.
- `delete-min()`/`delete-max()`.
- `iterator()`:
  Går igenom elementen i sorterad ordning.

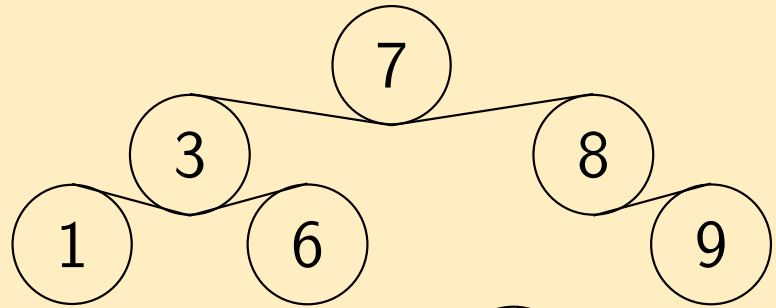
Hashtabell implementerar inte effektivt de tre sista.

Vilka träd är binära sökträd?

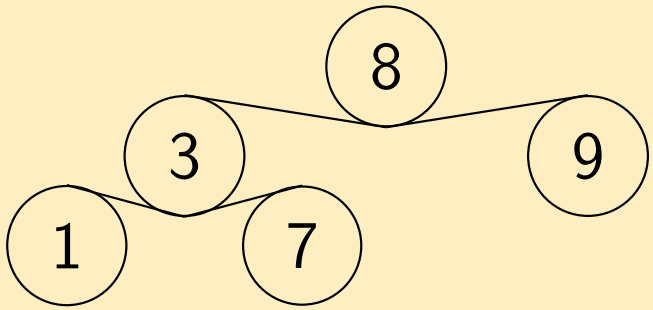
A:



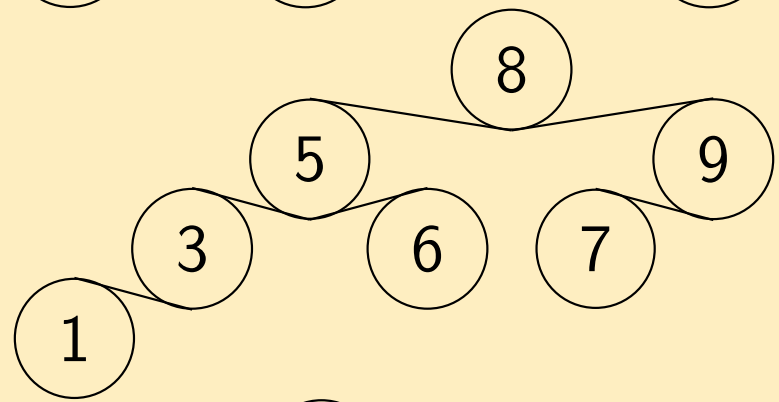
B:



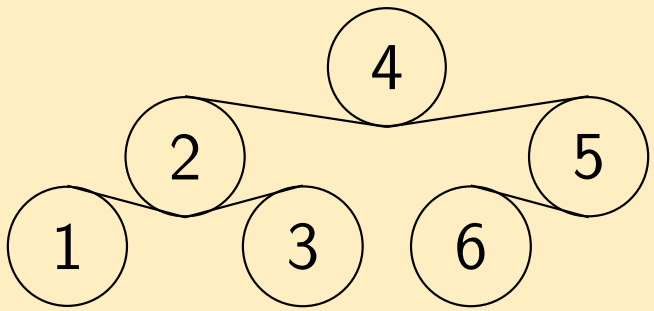
C:



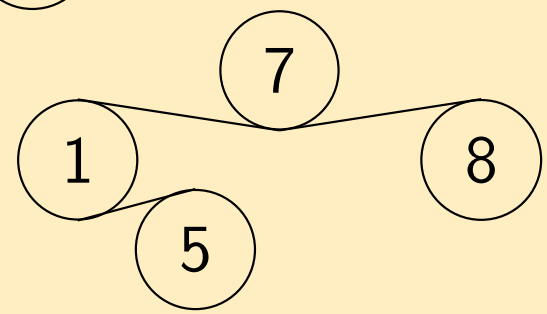
D:



E:

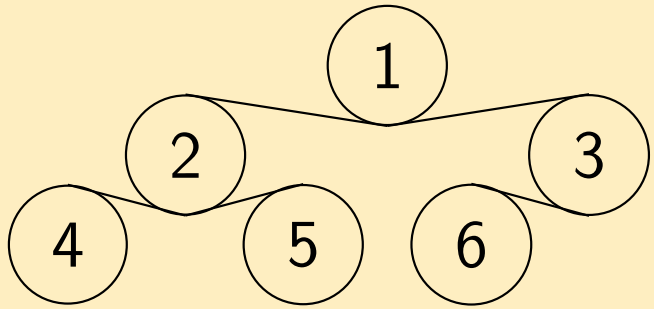


F:

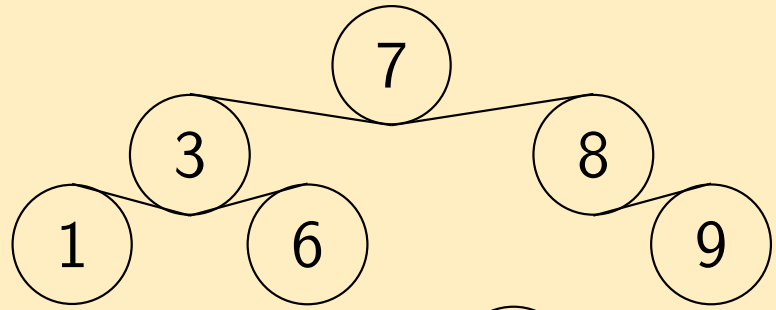


Vilka träd är binära sökträd?

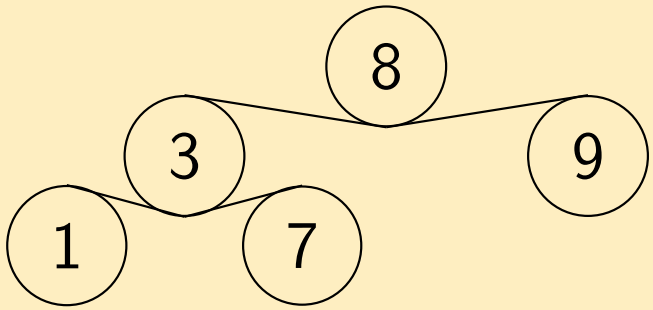
A:



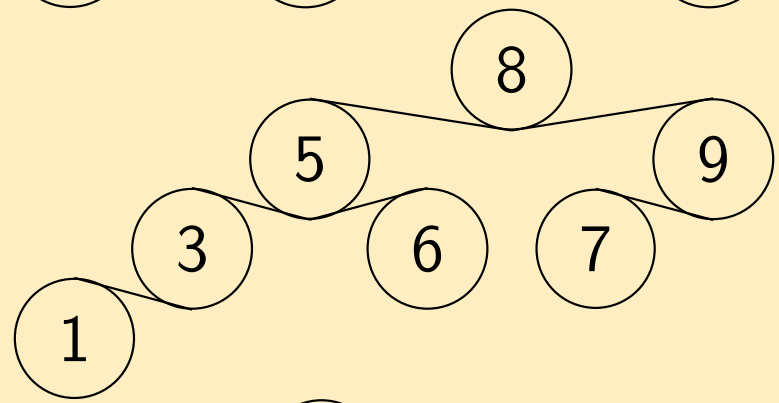
B:



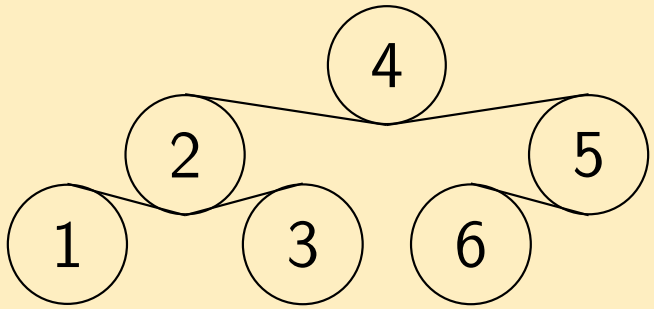
C:



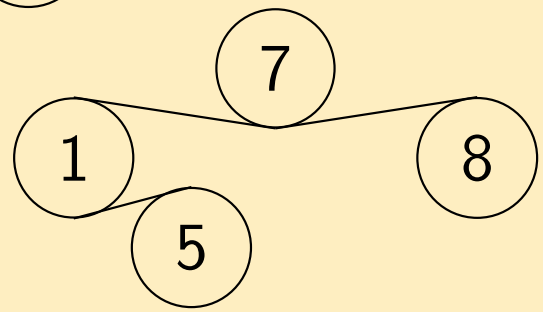
D:



E:



F:



Svar: B, C och F.

Obalanserade binära sökträd

En möjlig implementation:

```
public class BinarySearchTree
  <A extends Comparable<? super A>> {

  private class Node {
      A    contents;
      Node left;   // null om vänster barn saknas.
      Node right;  // null om höger barn saknas.

      Node(A contents) {
          this.contents = contents;
      }
  }

  private Node root;  // null om trädet är tomt.
```

Obalanserade binära sökträd

Iterativ kod:

```
public boolean member(A a) {
    Node here = root;

    while (here != null) {
        int cmp = a.compareTo(here.contents);
        if      (cmp < 0) { here = here.left; }
        else if (cmp > 0) { here = here.right; }
        else return true;
    }

    return false;
}
```

Obalanserade binära sökträd

Rekursiv kod (varning för stack overflow):

```
public void insert(A a) {
    root = insert(a, root);
}

private Node insert(A a, Node n) {
    if (n == null) return new Node(a);

    int cmp = a.compareTo(n.contents);
    if      (cmp < 0) n.left  = insert(a, n.left);
    else if (cmp > 0) n.right = insert(a, n.right);
    return n;
}
```

Obalanserade binära sökträd

Hur tar man bort ett element? Förslag:

- Hitta nod som ska tas bort (om någon).
- Lätt om noden har noll eller ett barn.
- Annars:
  Ta bort högra delträdets minsta elementet eller vänstra delträdets största element (dessa har max 1 barn).
  Ersätt elementet som ska tas bort med innehållet i den borttagna noden.

Obalanserade binära sökträd

Tidskomplexitet:

- `member`, `insert`, `delete`: $O(\textit{höjd})$.
- `deleteMin`: $O(\textit{höjd})$.
- inorder-genomlöpning: $\Theta(\textit{storlek})$.

Höjd:

- Värsta fallet: $\Theta(\textit{storlek})$.
- Värsta fallet uppstår t ex om man sätter in elementen 1, 2, 3, 4, ...

Kan vi se till att värstafallshöjden är $\Theta(\log \textit{storlek})$ (vilket gäller för kompletta träd, heapar)?

Balanserade sökträd

Balanserade sökträd

Sökträd som är balanserade,
med höjden $\Theta(\log \textit{storlek})$:

- AVL-träd (Adelson-Velsky & Landis).
- Röd-svarta träd (JDK: `TreeMap`).
- B^+-träd.
- ...

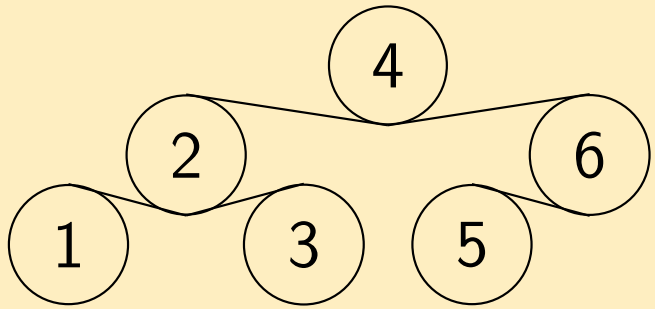
AVL-träd

AVL-träd

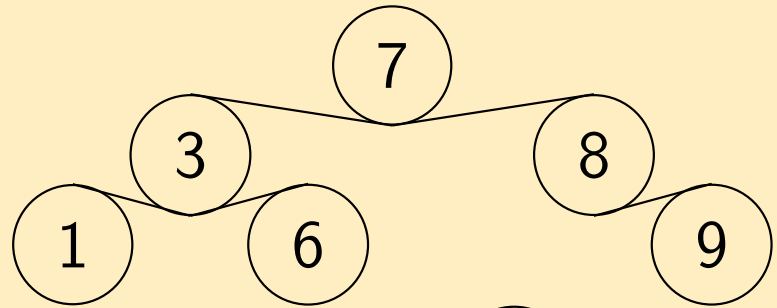
- Binärt sökträd.
- Invariant (för varje nod):
  Vänster och höger delträds höjder skiljer sig maximalt med 1.
- Höjd: $\Theta(\log n)$.
- Operationer som ändrar trädets struktur använder (ibland) *rotationer* för att återställa invarianten.

Vilka träd är AVL-träd?

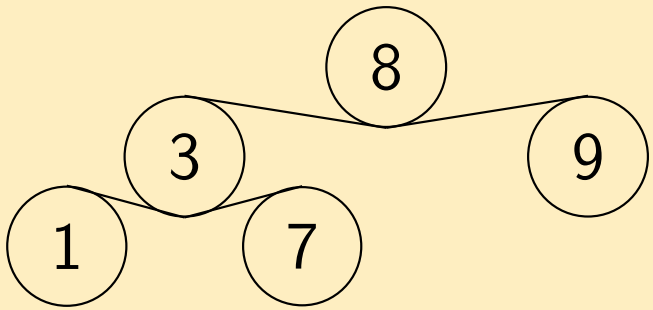
A:



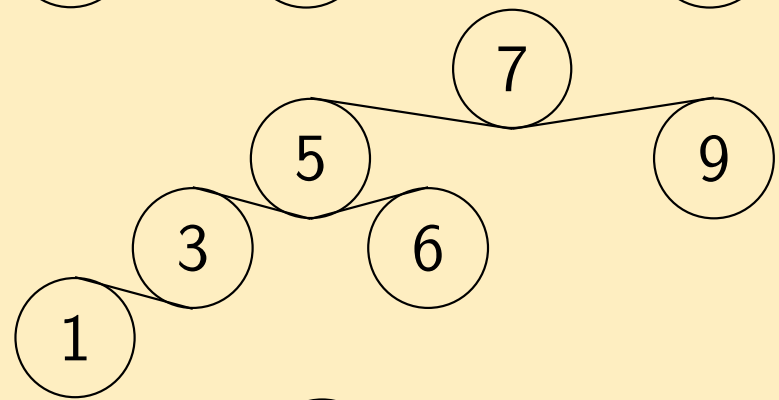
B:



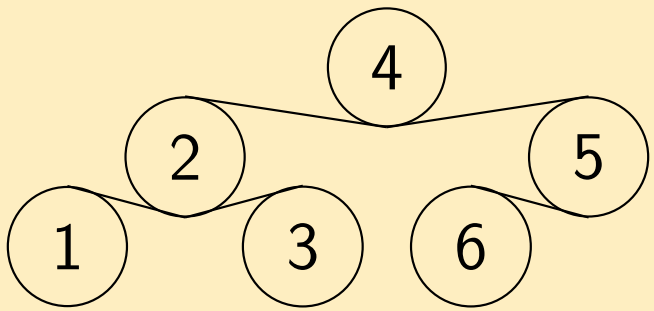
C:



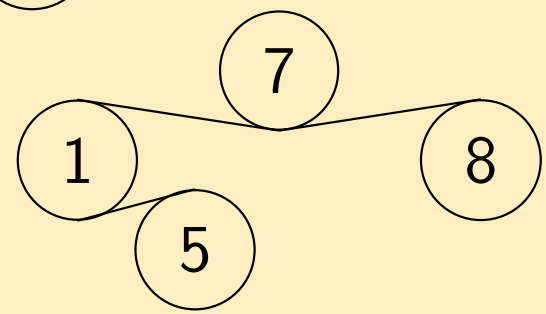
D:



E:

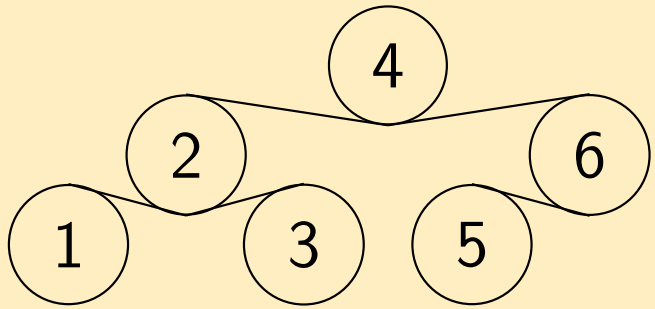


F:

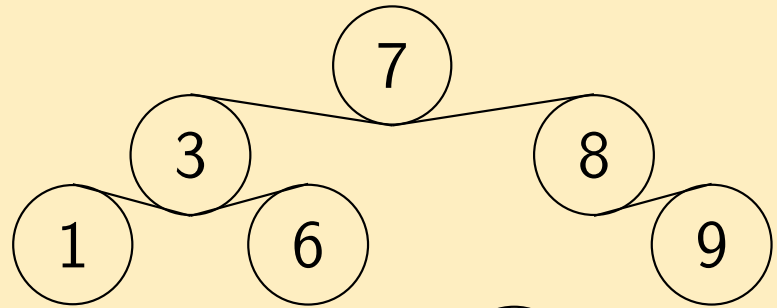


Vilka träd är AVL-träd?

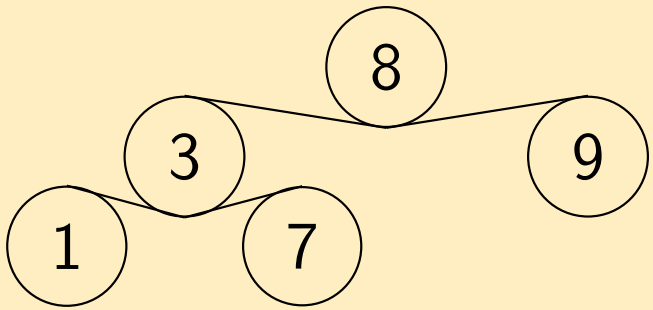
A:



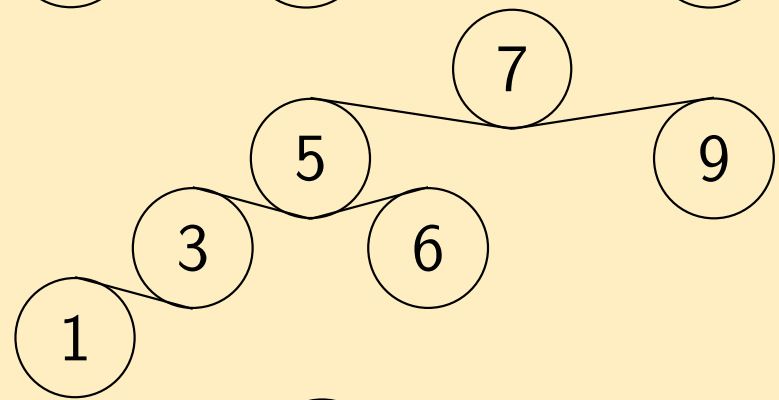
B:



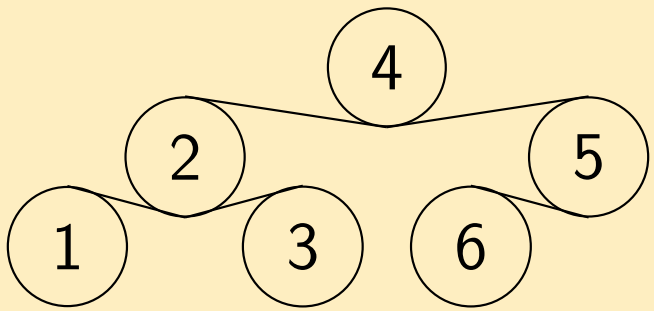
C:



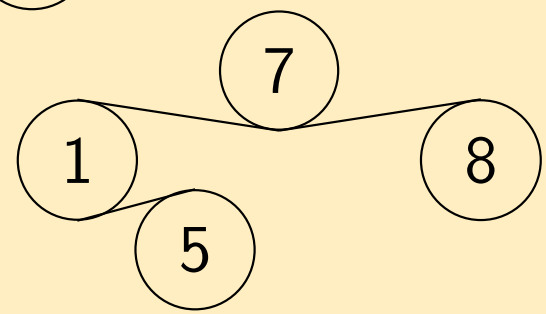
D:



E:



F:



Svar: A, B, C, F.

AVL-träd

Implementation:

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

member

Som för obalanserade binära sökträd.

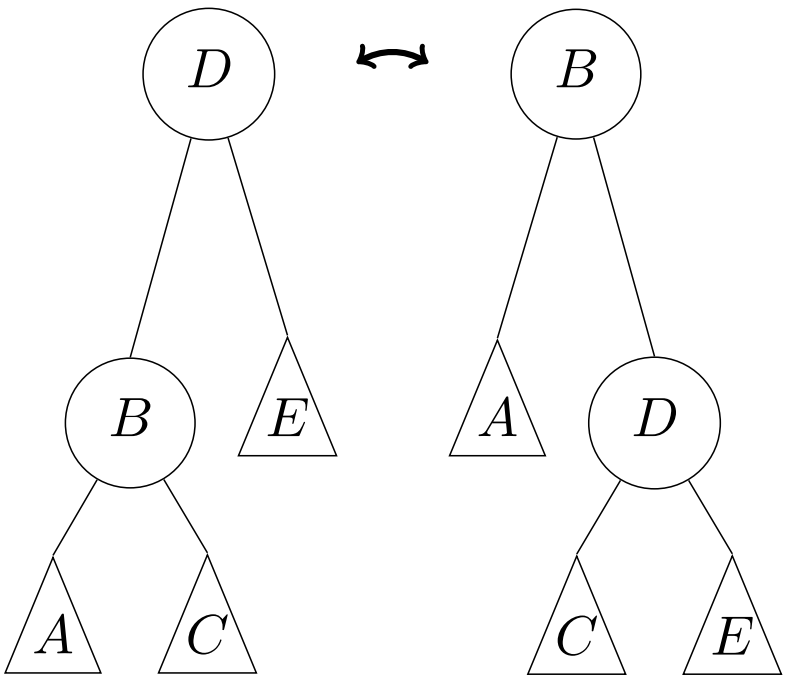
insert

Skiss av en algoritm:

- Sätt in noden som vanligt. Detta bryter ej sökträdsegenskapen, men kanske balanseringen.
- Gå tillbaka mot roten, uppdatera höjder.
- Vid första obalanserade noden: rotation.
- Antingen enkel- eller dubbelrotation.
- Det räcker med en rotation för att göra trädet balanserat.

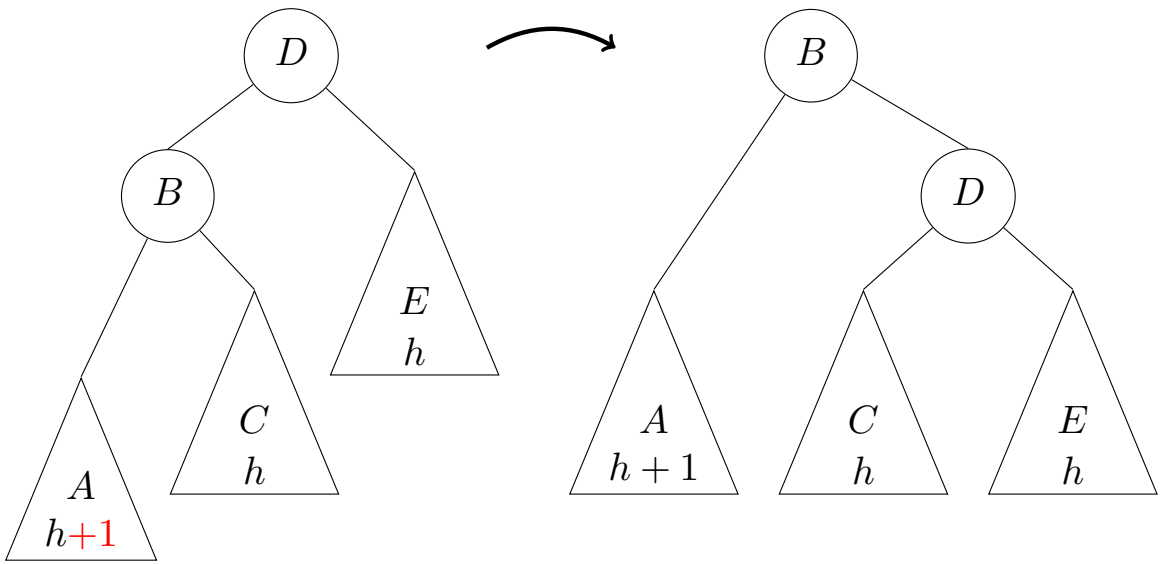
Trädrotationer

Grundläggande (enkel-) trädrotationer, höger- och vänster-. Ändrar strukturen, bevarar ordningen.



Enkelrotation, vänster-vänster- eller höger-höger-barn

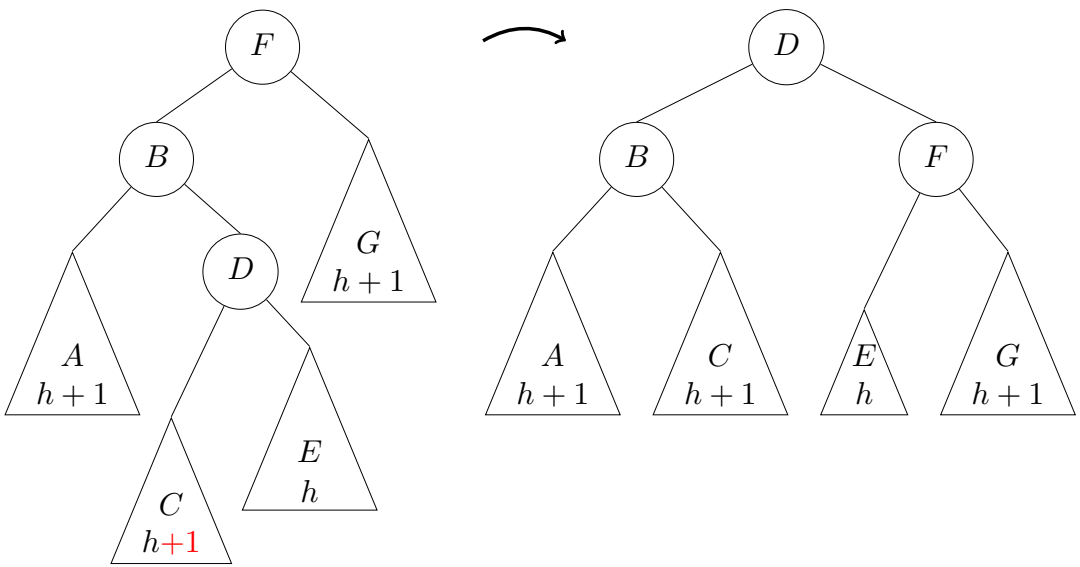
Om vi satte in en ny nod i A och dess höjd ökade (med 1), och första obalansen hittas i D:



Höjd innan insättning $= h + 2 =$ ny höjd.

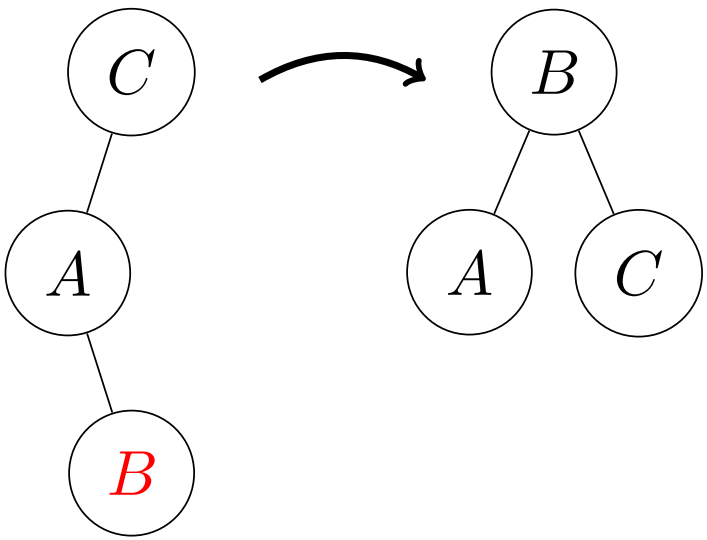
Dubbelrotation, vänster-höger- eller höger-vänster-barn

Om C:s höjd ökade med ett och första obalansen hittas i F: (Motsvarande om E:s höjd ökade.)



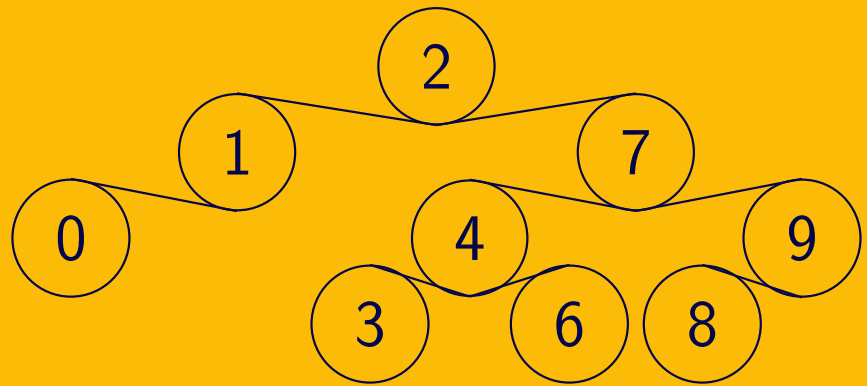
Höjd innan insättning $= h + 3 =$ ny höjd.
Två enkelrotationer.

Dubbelrotation, specialfall

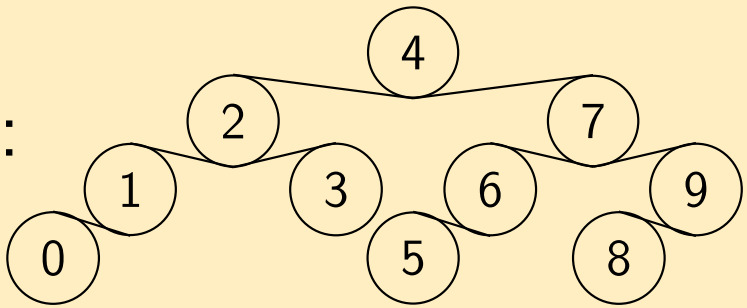


Höjd innan insättning $= 1 =$ ny höjd.
Två enkelrotationer.

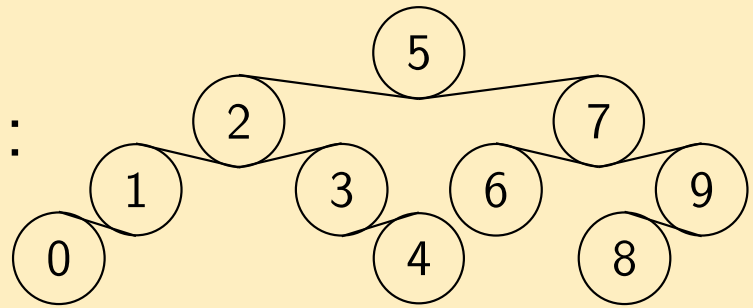
Vad blir resultatet av att sätta in 5 i följande AVL-träd?



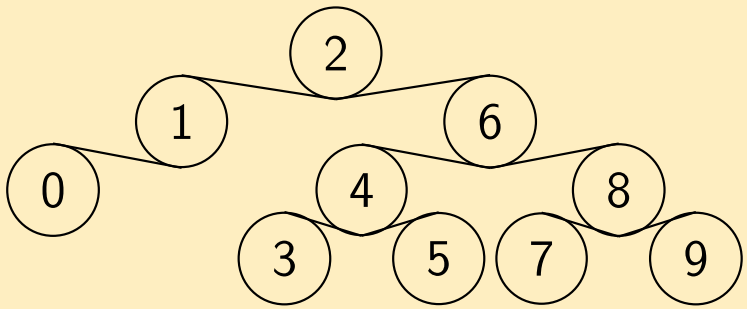
A:



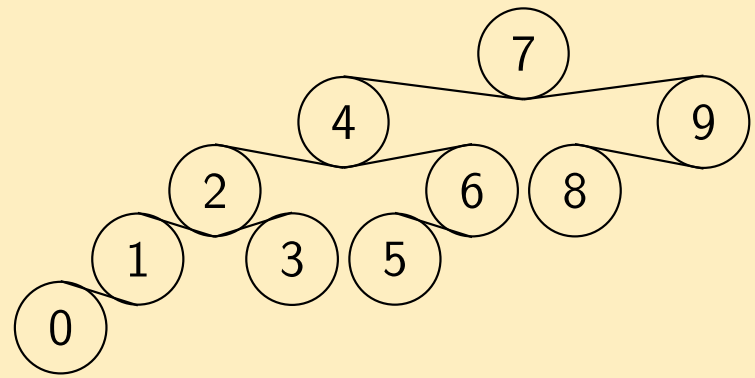
B:



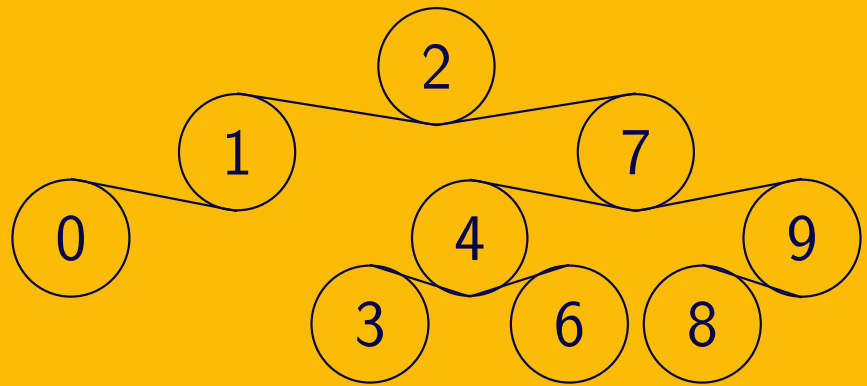
C:



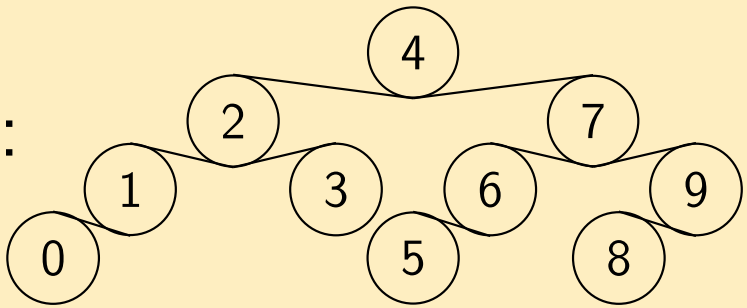
D:



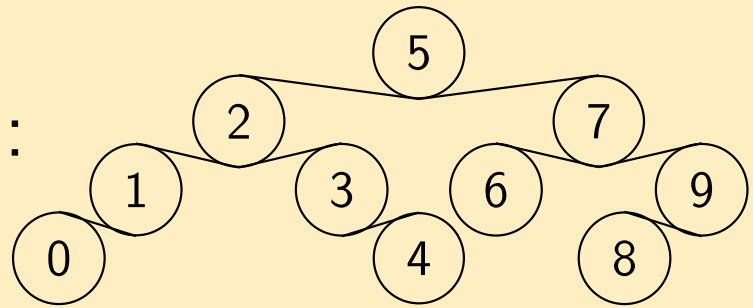
Vad blir resultatet av att sätta in 5 i följande AVL-träd?



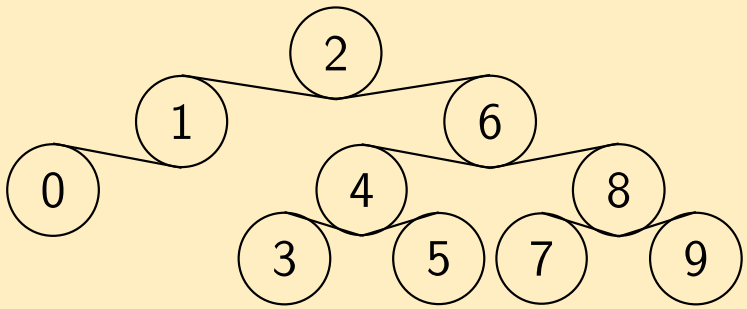
A:



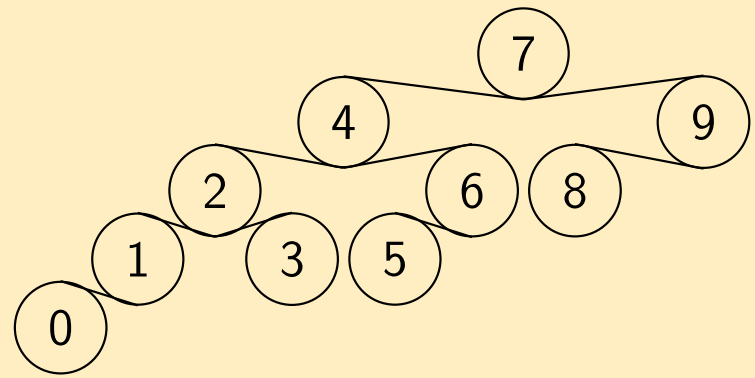
B:



C:



D:



Svar: A

delete

- Kan utgå från algoritmen för obalanserade binära sökträd.
- Vandra mot roten (från noden som togs bort, vilket inte alltid där elementet fanns), uppdatera höjder och titta efter obalans på samma sätt som för insättning.
- Vid obalans utför liknande rotationer som för insättning.
- Alla rotationer gör vid borttagning inte att delträdets höjd förblir oförändrat. Man får därför ibland fortsätta uppåt även efter en rotation.

AVL-träd, komplexitet

- Eftersom höjden är $\Theta(\log n)$ så är tar alla operationer $O(\log n)$.
- Genomlöpning i ordning tar $\Theta(n)$.

AVL-träd

Animerat: `https://www.cs.usfca.edu/~galles/visualization/AVLtree.html`.

Röd-svarta träd

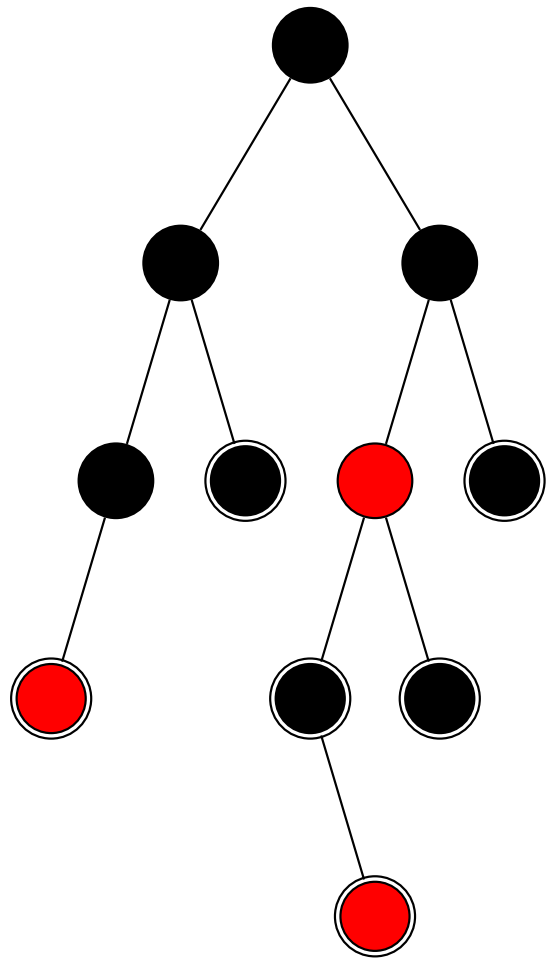
Röd-svarta träd

- JDK 8: `TreeSet`, `TreeMap`.
- Höjd: $\Theta(\log n)$. Samma tidskomplexitet för operationerna som för AVL-träd.

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.
- `https://www.cs.usfca.edu/~galles/visualization/RedBlack.html`



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 nod.
- Gör noder som ska sparas på disk så stora att de fyller ett diskblock (när de är fulla).
- Genom att kräva att antalet barn till interna noder är minst $M/2$ blir upprätthållandet av formen ganska enkel och den övre gränsen för djupet bra.
- Används i filsystem och databaser.