Föreläsning 9
Datastrukturer (DAT037)

Fredrik Lindblad[1]

2015-11-28

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

Idag

- Sökträd.
  - Obalanserade
  - Balanserade – AVL

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 (oftast?)
  implementera en *strikt total ordning*:
  - Om $x < y$ och $y < z$ så är $x < z$.
  - Exakt en av följande gäller:
    $x < y$, $x = y$, $x > y$.

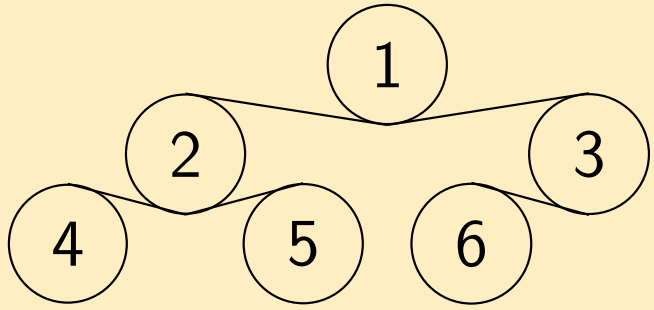
Binära sökträd

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

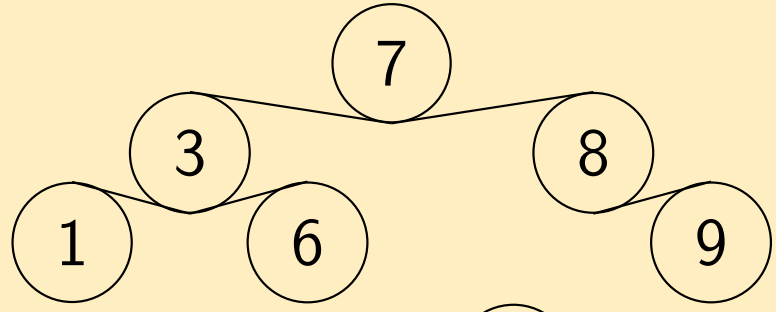
- Konstruerare: Tom mängd/avbildning.
- `member(k)`/`lookup(k)`.
- `insert(k)`/`insert(k,v)`.
- `delete(k)`.
- `find-min()`/`find-max()`.
- `delete-min()`/`delete-max()`.
- `iterator()`:
  Går igenom elementen i sorterad ordning.

Vilka träd är binära sökträd?

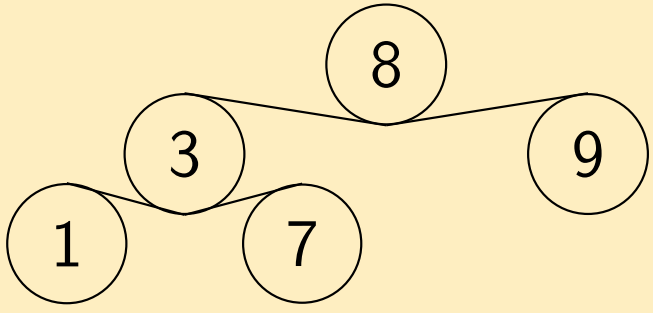
A:



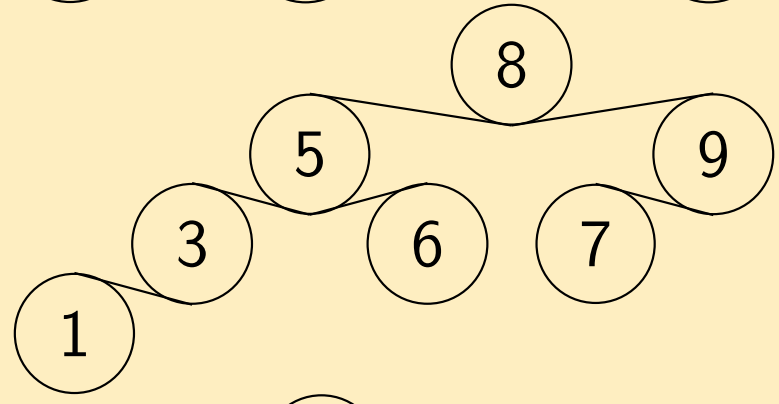
B:



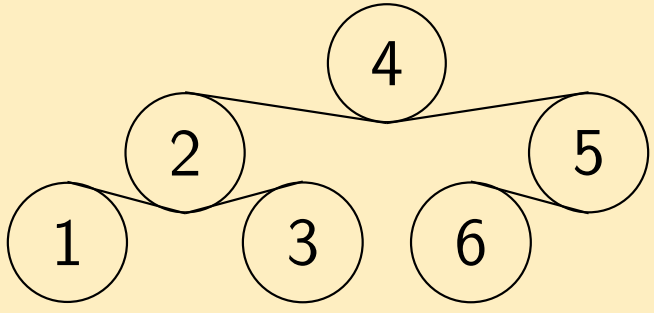
C:



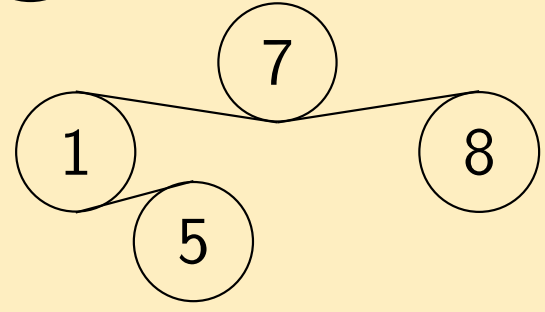
D:



E:



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);
    else n.contents = a;  // Skriver över.
    return n;
}
```

Obalanserade binära sökträd

Hur tar man bort ett element? Förslag:

- Hitta nod som ska tas bort (om någon).
- Lätt om noden bara har ett barn.
- Annars:
  Ta bort högra delträdets minsta elementet
  eller vänstra delträdets största element
  (dessa har max 1 barn),
  använd som nodens innehåll.

Alternativ: Lat borttagning.

Obalanserade binära sökträd

Tidskomplexitet:

- `inorder`: $\Theta(\textit{storlek})$.
- `deleteMin`: $O(\textit{höjd})$.
- `member`, `insert`, `delete`:
  $O(\textit{höjd})$, givet att jämförelser tar konstant tid.

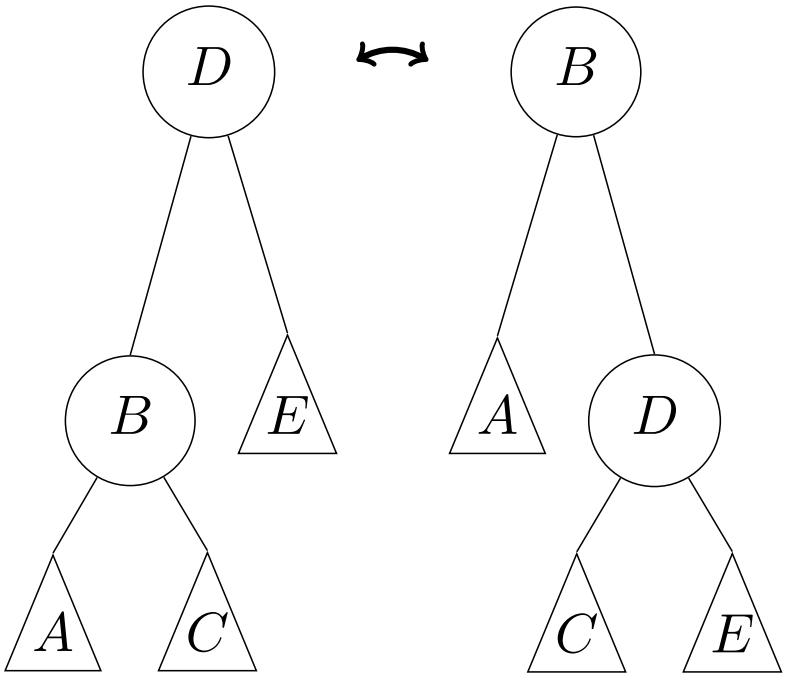
Höjd:

- 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 $\Theta(\log \textit{storlek})$?

Trädrotationer

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



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.
- ...

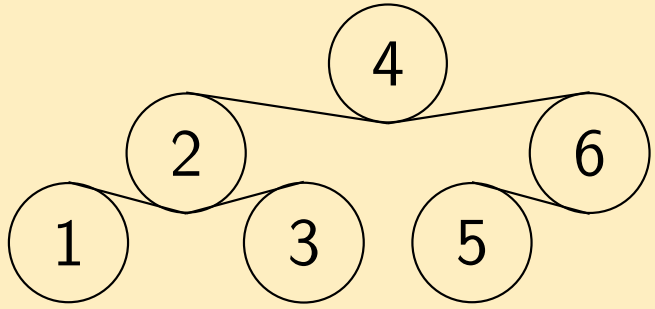
AVL-träd

AVL-träd

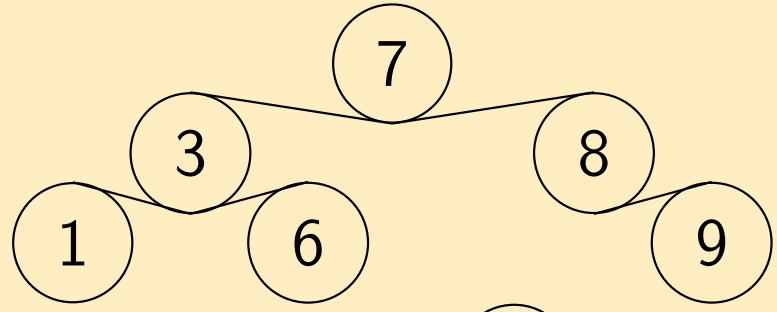
- Binärt sökträd.
- Invariant (för varje nod):
  Vänster och höger delträd har samma höjd, ± 1.
- Höjd: $\Theta(\log n)$.
- Operationer som ändrar trädets struktur använder (ibland) *rotationer* för att återställa invarianten.

Vilka träd är AVL-träd?

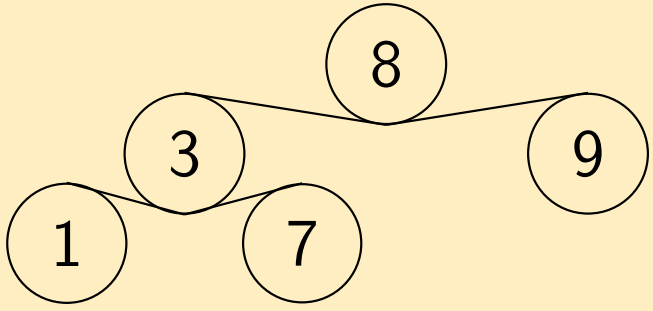
A:



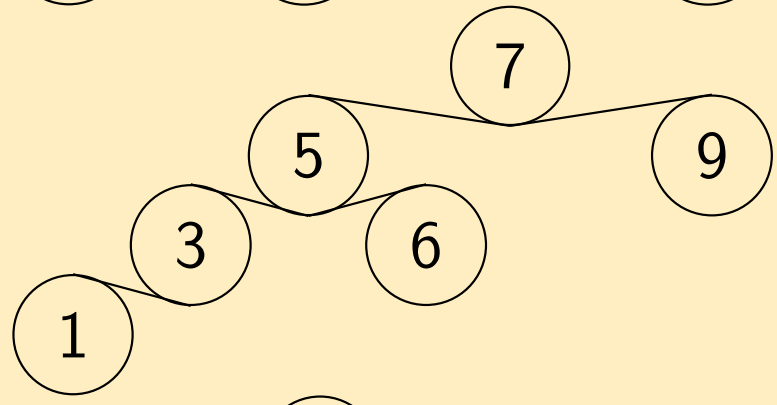
B:



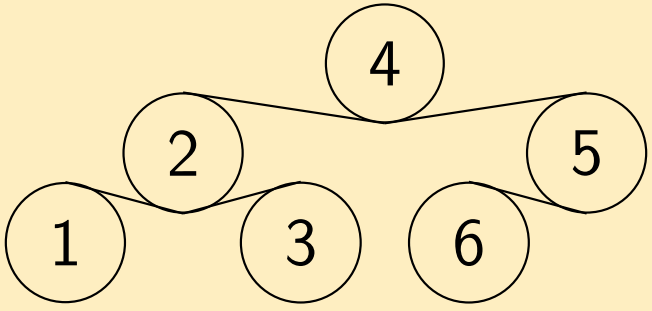
C:



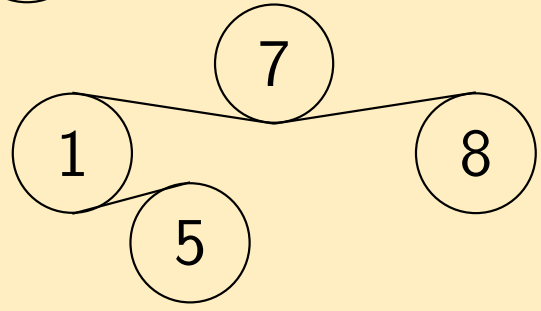
D:



E:



F:



AVL-träd

Implementation:

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

member

Som för obalanserade binära sökträd.

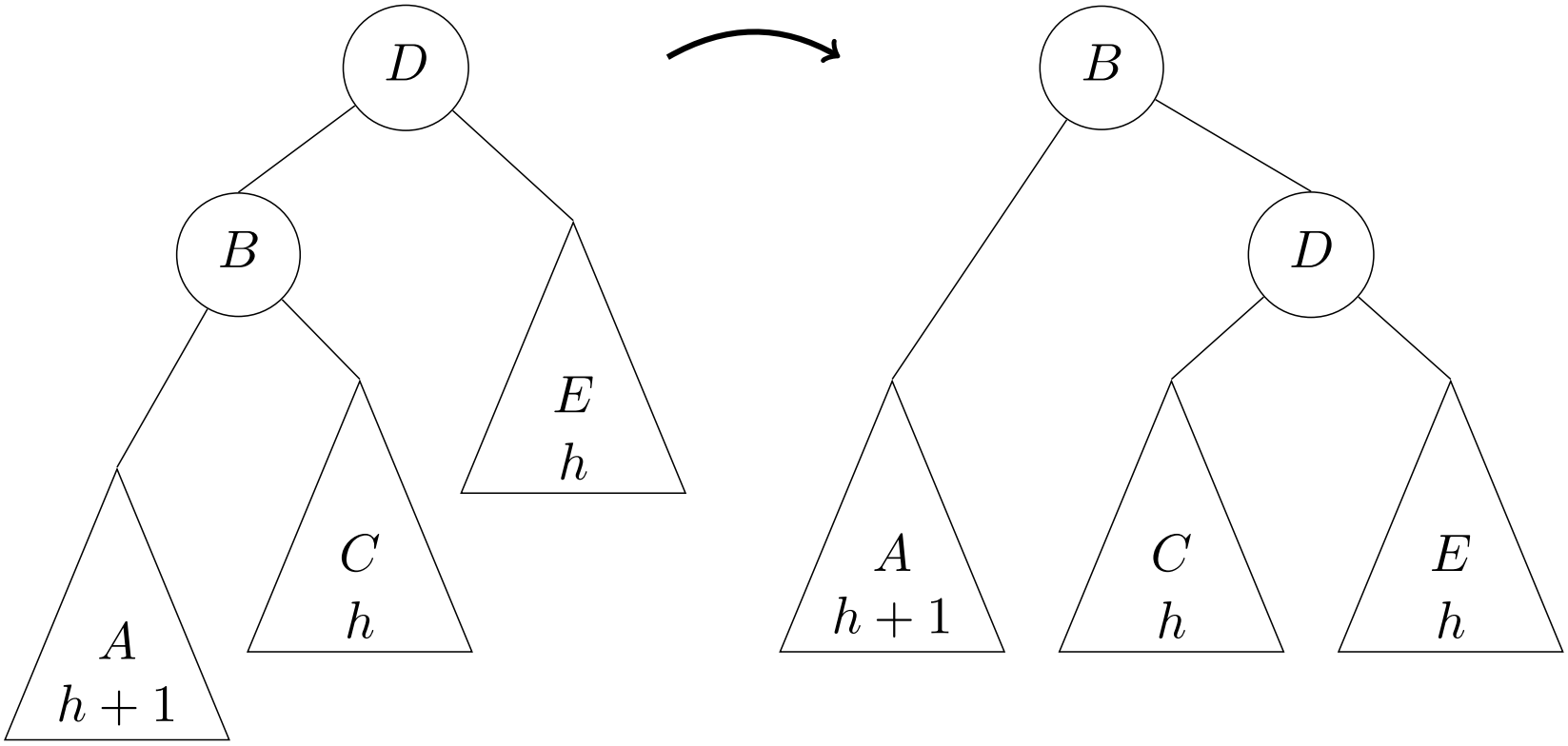
`insert`

Skiss av en algoritm:

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

Enkelrotation

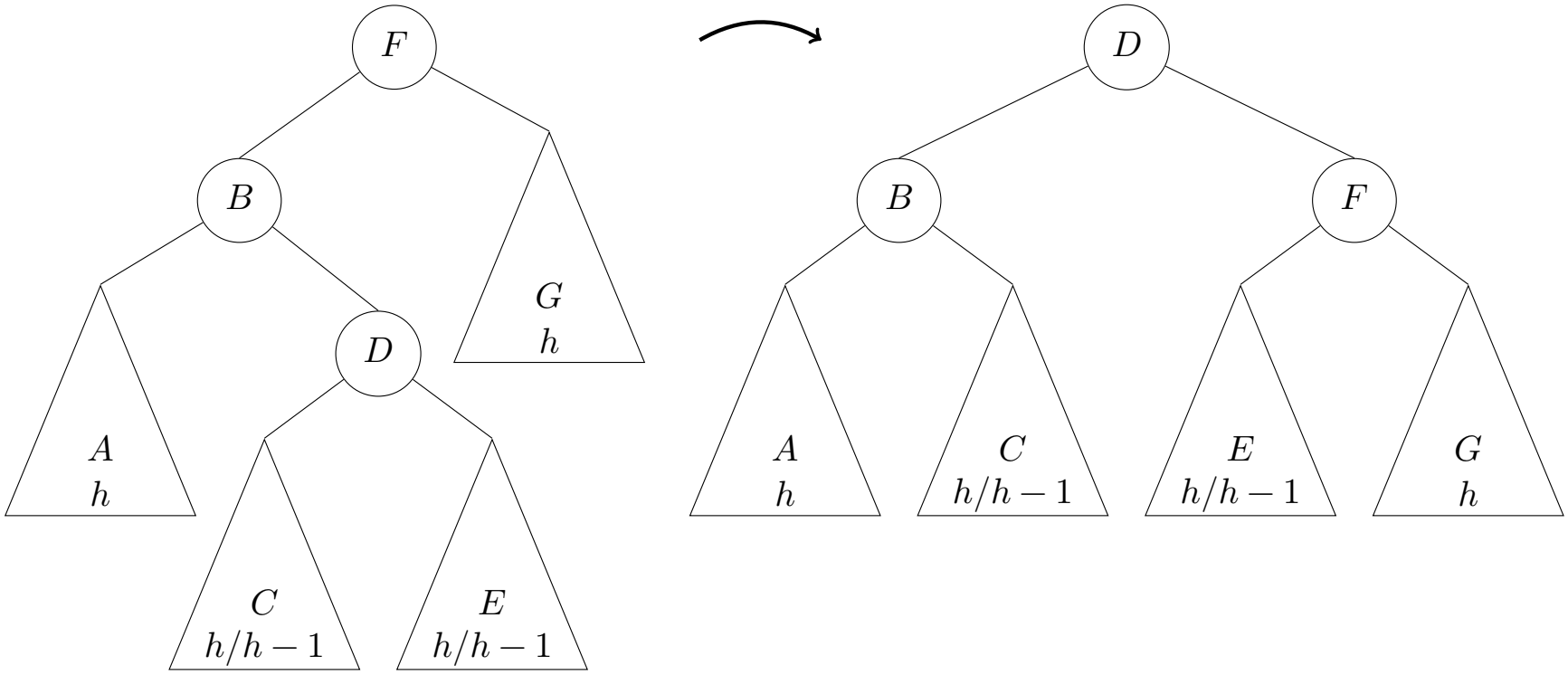
Om vi satte in en ny nod i A,
och första obalansen hittas i D:



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

Dubbelrotation

Om vi satte in en ny nod i $C/D/E$,
och första obalansen hittas i F:



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

`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`.