# Föreläsning
# Datastrukturer (DAT037)

Nils Anders Danielsson

2015-11-30

# Idag

- Sökträd.
  - Obalanserade.
  - Balanserade.
- Prefixträd (om vi hinner).

# 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.

# 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:
    $x < y$, $x = y$, $x > y$.

# Komparator i Haskell

Komparator för typen `a`:

```
data Ordering = LT | EQ | GT

a -> a -> Ordering
```

Typklassen `Ord` innehåller `compare`:

```
compare :: Ord a => a -> a -> Ordering

compare 2 3 = LT
compare 2 2 = EQ
compare 3 2 = GT
```

# 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.

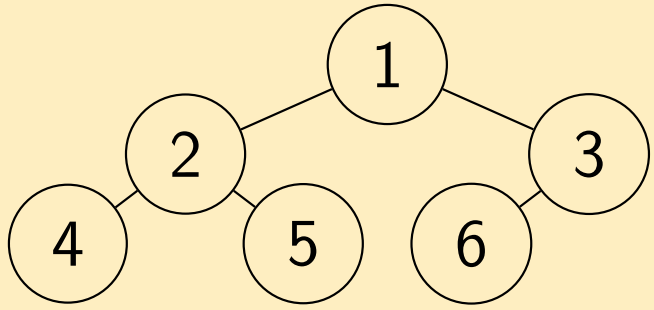
# Binära sökträd

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

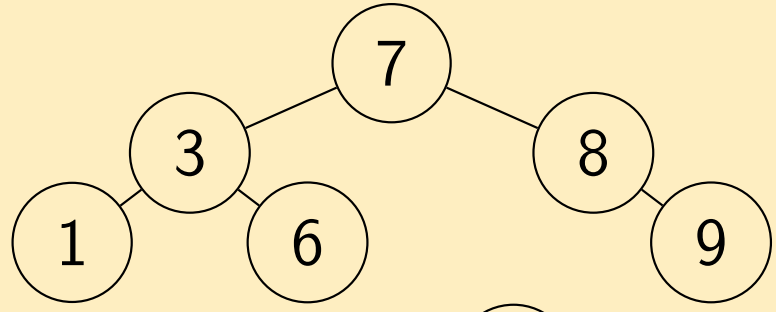
- `empty`: Tom mängd/avbildning.
- `member k t`/`lookup k t`.
- `insert k t`/`insert k v t`.
- `delete k t`.
- `delete-min t`/`delete-max t`.
- `inorder t`:
  En sorterad lista med alla element.

# Vilka träd är binära sökträd?

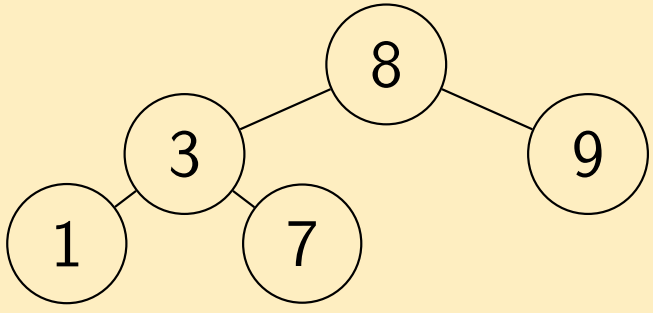
A:



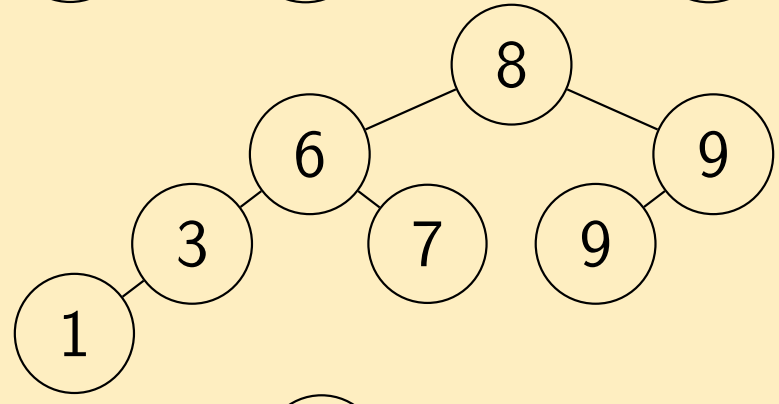

B:



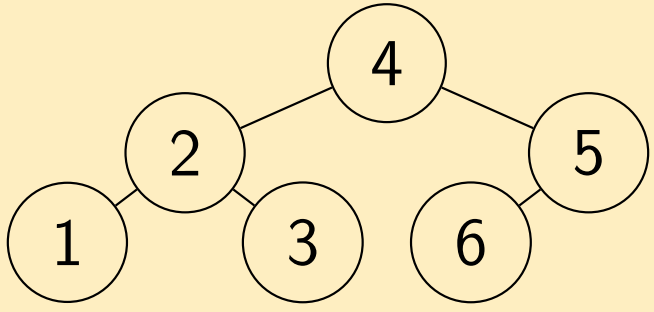

C:



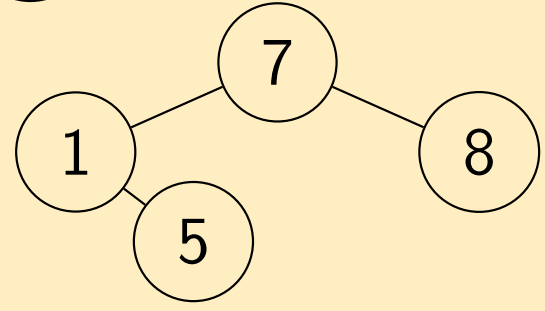

D:




E:




F:



# Obalanserade binära sökträd

En möjlig implementation:

```
module BinarySearchTree
  (Tree, empty, member, insert,
   deleteMin, delete, inorder)
  where

data Tree a = Empty
            | Node (Tree a) a (Tree a)

empty :: Tree a
empty = Empty
```

# Obalanserade binära sökträd

```
data Tree a = Empty
            | Node (Tree a) a (Tree a)

-- Kan söka rekursivt:
member :: Ord a => a -> Tree a -> Bool
member x Empty        = False
member x (Node l y r) = case compare x y of
  LT -> member x l
  EQ -> True
  GT -> member x r
```

# Obalanserade binära sökträd

```
-- Sök, stoppa in.
insert :: Ord a => a -> Tree a -> Tree a
insert x Empty          = Node Empty x Empty
insert x (Node l y r) = case compare x y of
  LT -> Node (insert x l) y r
  EQ -> Node l x r                -- Skriver över.
  GT -> Node l y (insert x r)
```

# Obalanserade binära sökträd

```
-- Minsta värdet finns längst till vänster
-- (om trädet inte är tomt).
deleteMin :: Tree a -> Maybe (a, Tree a)
deleteMin Empty          = Nothing
deleteMin (Node l x r) = case deleteMin l of
  Nothing         -> Just (x, r)
  Just (min, l') -> Just (min, Node l' x r)
```

# Obalanserade binära sökträd

Hur tar man bort ett element? Förslag:

- Hitta nod som ska tas bort (om någon).
- Lätt om noden inte har något högerbarn.
- Annars:
  Ta bort högra delträdets minsta elementet,
  använd som nodens innehåll.

Alternativ: Lat borttagning.

# Obalanserade binära sökträd

```
delete :: Ord a => a -> Tree a -> Tree a
delete x Empty          = Empty
delete x (Node l y r) = case compare x y of
  LT -> Node (delete x l) y r
  GT -> Node l y (delete x r)
  EQ -> case deleteMin r of
    Nothing         -> l
    Just (min, r') -> Node l min r'
```

# Obalanserade binära sökträd

```
-- Går igenom trädet i inordning.
-- Postcondition: Listan är sorterad.
inorder :: Tree a -> [a]
inorder Empty        = []
inorder (Node l x r) = inorder l ++ (x : inorder r)
```

# Obalanserade binära sökträd

```
-- Går igenom trädet i inordning.
-- Postcondition: Listan är sorterad.
inorder :: Tree a -> [a]
inorder Empty        = []
inorder (Node l x r) = inorder l ++ (x : inorder r)
```

Tidskomplexitet för `xs ++ ys`: $\Theta(|\texttt{xs}|)$.

## Vad är värstafallstidskomplexiteten för `inorder t` (om `t` innehåller $n$ element)?

- $\Theta(1)$.
- $\Theta(\log n)$.
- $\Theta(n)$.
- $\Theta(n \log n)$.
- $\Theta(n^2)$.
- $\Theta(n^2 \log n)$.
- $\Theta(n^3)$.

# Obalanserade binära sökträd

Mer effektivt med ackumulator,
`(:)` har tidskomplexiteten $\Theta(1)$:

```
inorder :: Tree a -> [a]
inorder t = inorder' t []
  where
  inorder' :: Tree a -> [a] -> [a]
  inorder' Empty        xs = xs
  inorder' (Node l x r) xs =
    inorder' l (x : inorder' r xs)
```

Notera att

$$\texttt{inorder' t xs} = \texttt{inorder}_{\texttt{långsam}}\ \texttt{t ++ xs}.$$

# 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

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

# Balanserade sökträd

# Balanserade sökträd

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

- AVL-träd (Adelson-Velsky & Landis/
  Адельсóн-Вéльский & Лáндис).
- Röd-svarta träd (JDK: `TreeMap`).
- Storleksbalanserade träd (Haskell: `Data.Map`).
- $B^+$-träd.
- ...

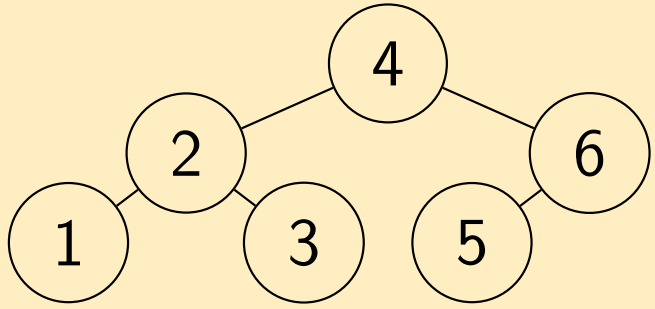
# AVL-träd

# AVL-träd

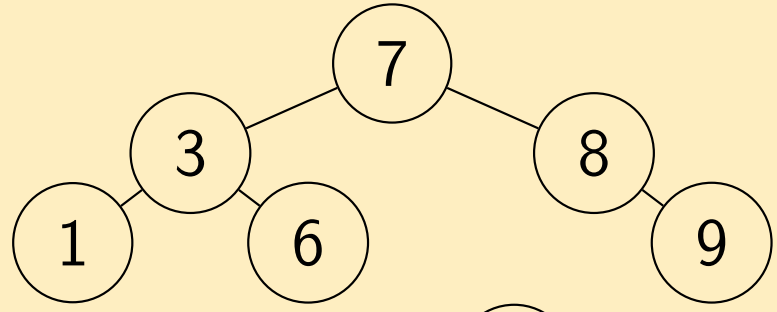
- Binärt sökträd.
- Invariant (för varje nod):
  Vänster och höger delträd har samma höjd, $\pm 1$.
- Höjd: $\Theta(\log n)$.
- Operationer som ändrar trädets struktur
  använder (ibland) *rotationer* för att
  återställa invarianten.

# Vilka träd är AVL-träd?

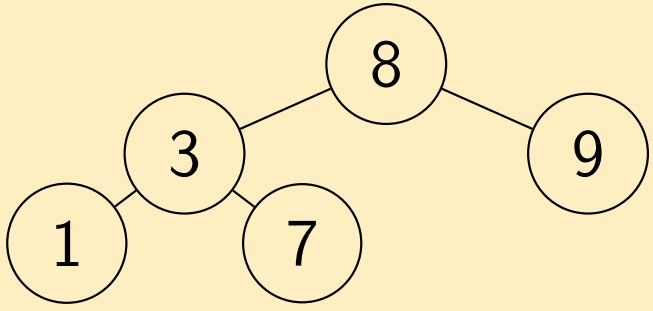
A:



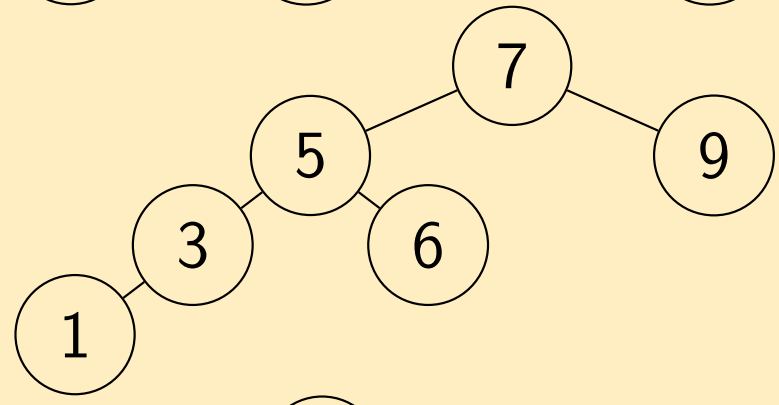

B:



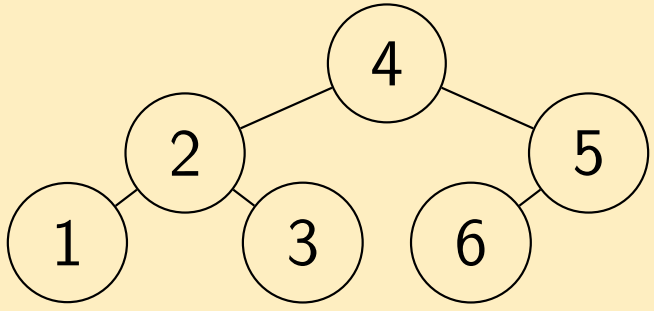

C:



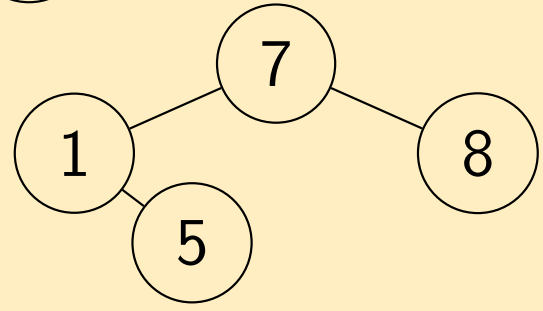

D:




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.

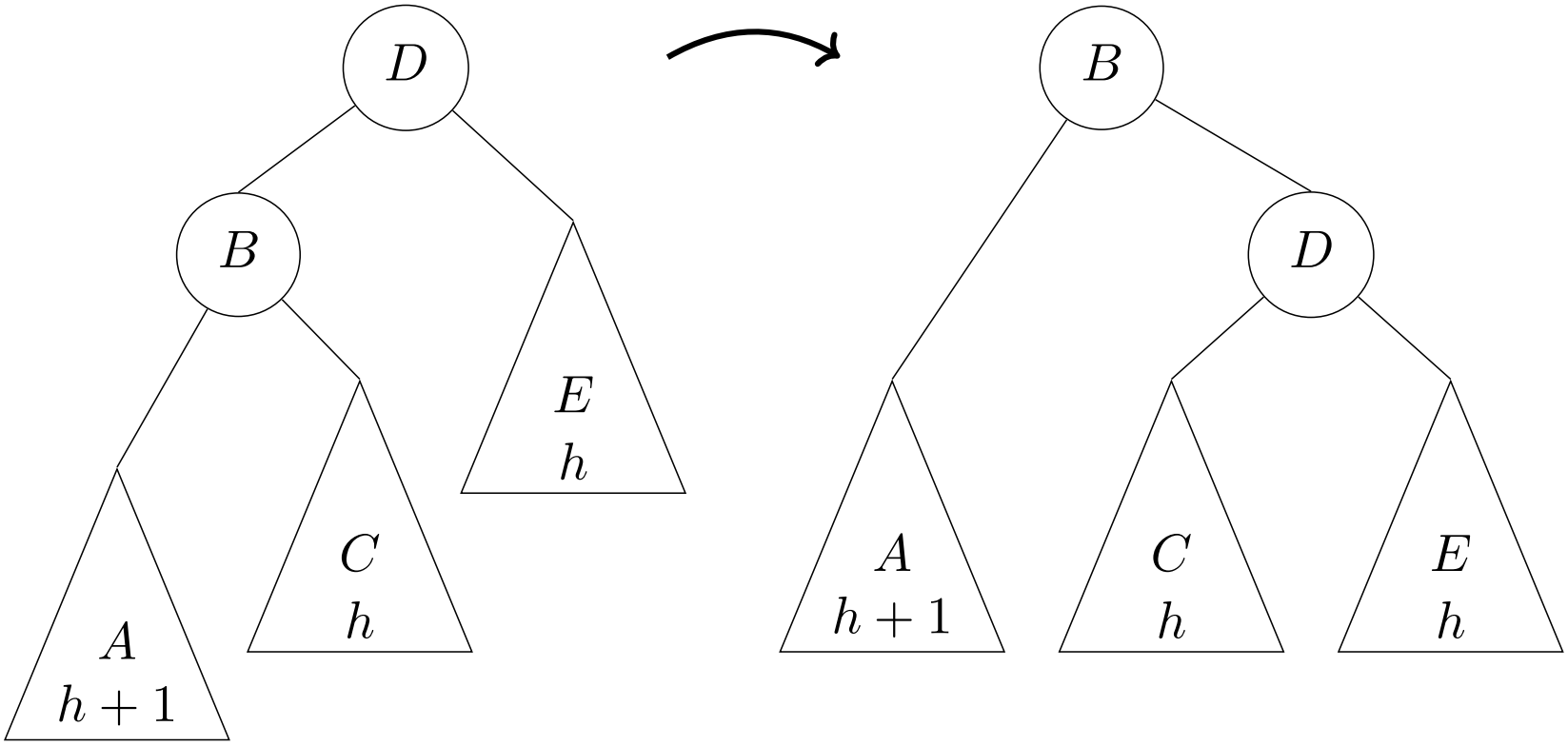
# `insert`

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

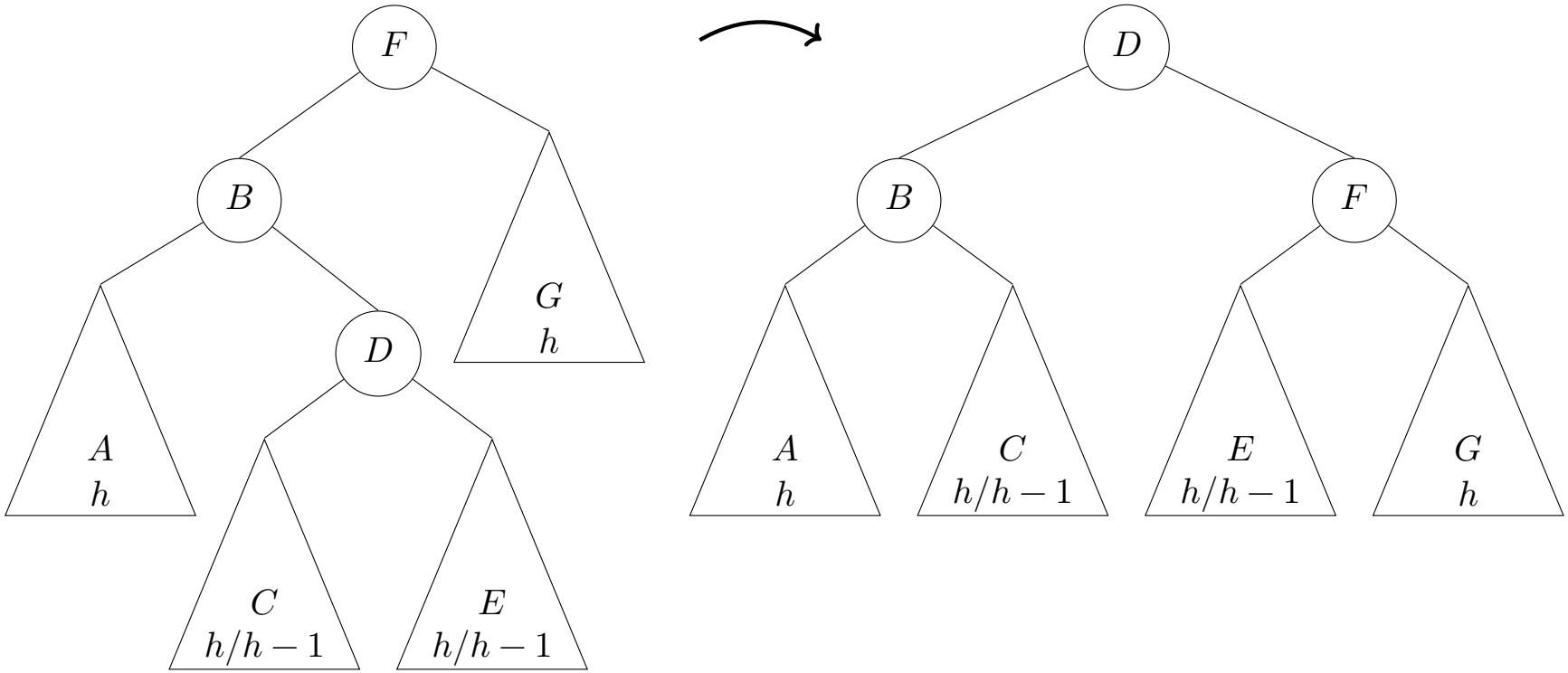
Om vi satte in en ny nod i $A$,
och första obalansen hittas i $D$:




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.

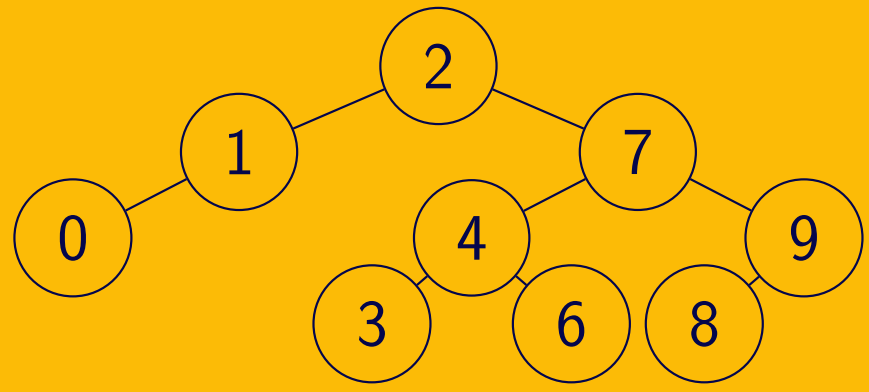
# `delete`

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

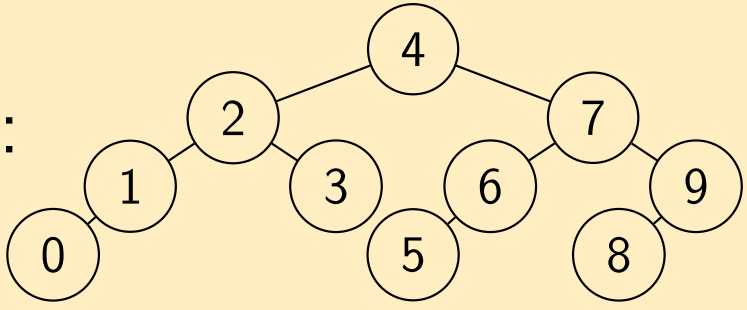
# AVL-träd

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

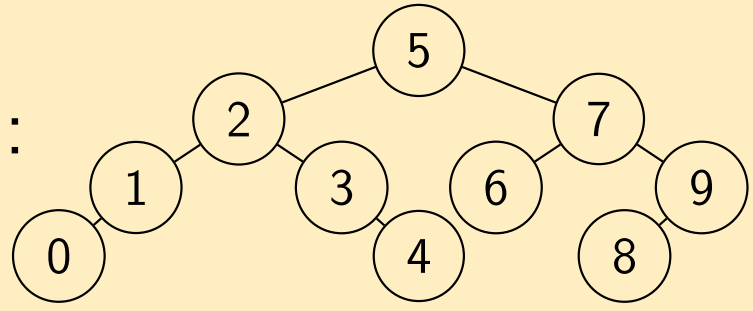
Vad blir resultatet av att sätta in 5 i följande AVL-träd?



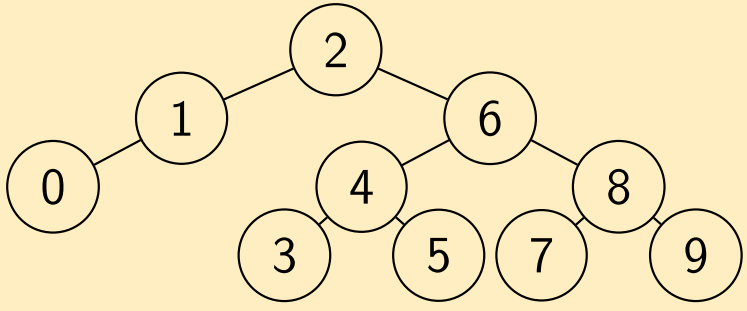

A:



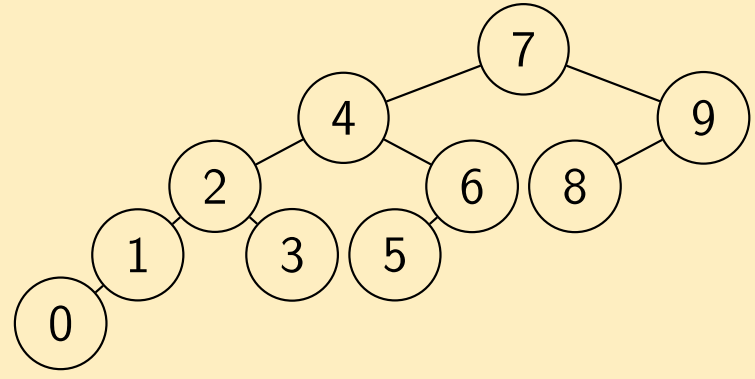

B:




C:




D:



# Röd-svarta träd

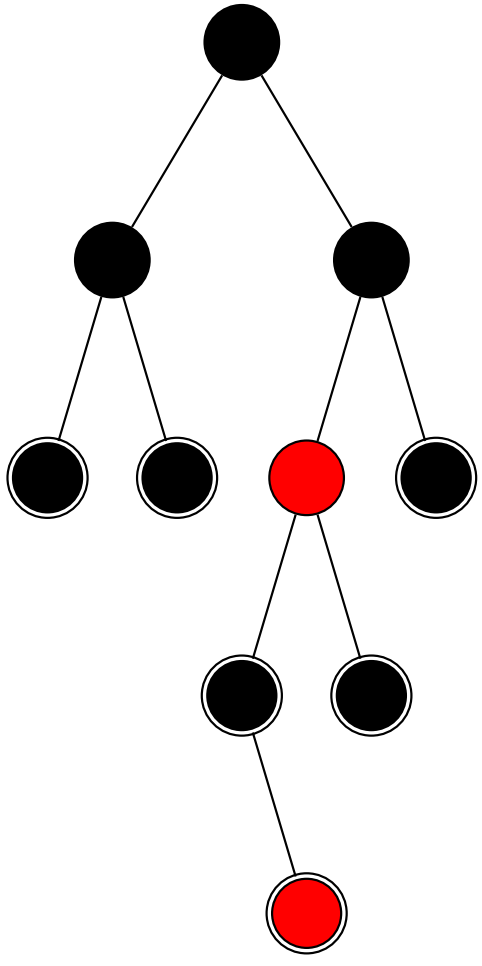
# Röd-svarta träd

- JDK 8: `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.



# 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.

# Prefixträd

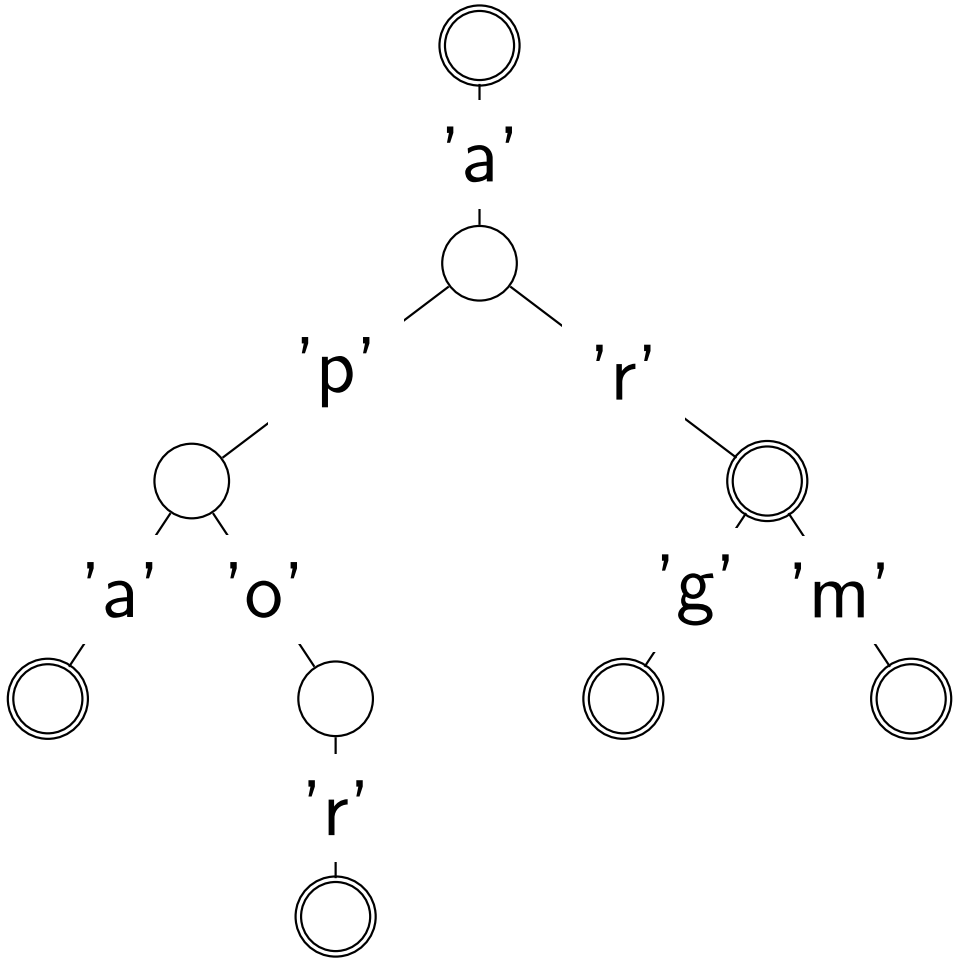
# Prefixträd (tries)

- Kan implementera mängder och avbildningar.
- Nycklar: strängar.

# Prefixträd (tries)

{ "", "apa", "apor", "ar", "arg", "arm" }:

- Träd med tecken på kanterna.
- Noder:
  - Medlem?
  - Värde?



# Prefixträd (tries)

Mängd:

```
data Trie k =
  Node Bool (Map k (Trie k))
```

Några operationer:

```
empty  :: Trie k
member :: [k] -> Trie k -> Bool
insert :: [k] -> Trie k -> Trie k
delete :: [k] -> Trie k -> Trie k
toList :: Trie k -> [[k]]
```

Om trädet gås igenom i preordning:
`toList t` är en lexikografiskt sorterad lista.

# Prefixträd (tries)

Avbildning:

```
data Trie k v =
  Node (Maybe v) (Map k (Trie k v))
```

Några operationer:

```
empty  :: Trie k v
lookup :: [k] -> Trie k v -> Maybe v
insert :: [k] -> v -> Trie k v -> Trie k v
delete :: [k] -> Trie k v -> Trie k v
toList :: Trie k v -> [([k], v)]
```

# Prefixträd (tries)

Några alternativ för `Map`:

| | Array | Enkellänkad lista | AVL-träd |
|---|---|---|---|
| Storlek | $\Theta(\lvert\Sigma\rvert)$ | $\Theta(c)$ | $\Theta(c)$ |
| Sökning | $\Theta(1)$ | $O(c)$ | $O(\log c)$ |
| Insättning | $O(\lvert\Sigma\rvert)$ | $O(c)$ | $O(\log c)$ |

- $\Sigma$: Alfabetet $(0, 1, ..., |\Sigma| - 1)$.
- $c$: Antalet barn $(c \leq |\Sigma|)$.

Vad är värstafallstidskomplexiteten för att testa medlemskap av en sträng i en mängd som innehåller $n$ strängar? Varje sträng har längd $\ell$, prefixträdet använder arrayer, och hashfunktionen har tidskomplexiteten $\Theta(\ell)$.

- Hashtabell: $\Theta(\ell)$.
- Hashtabell: $\Theta(\ell n)$.
- Prefixträd: $\Theta(\ell)$.
- Prefixträd: $\Theta(\ell n)$.
- AVL-träd: $\Theta(\log n)$.
- AVL-träd: $\Theta(\ell \log n)$.

# Prefixträd (tries)

- Det finns flera varianter av prefixträd.
- En variant: HAMTs (hash array mapped tries).
- I Haskell: `HashMap` i `unordered-containers`.
- Nycklar: Hashkoder som ses som strängar av små tal, kanske 4 bitar i varje.
- Risk för kollisioner.
- Komprimerade arrayer, utan tomma delträd:
  - Bitmap visar vilka barn som finns.
  - Processorinstruktion för Hammingvikt kan göra datastrukturen mer effektiv.
- Kan göras persistent.

# Sammanfattning

Idag:

- Balanserade sökträd.
- Prefixträd (kanske).

Nästa gång:

- Duggan lämnas ut.
- Prefixträd (kanske).
- Skipplistor.