Föreläsning
Datastrukturer (DAT036)

Nils Anders Danielsson

2013-11-25

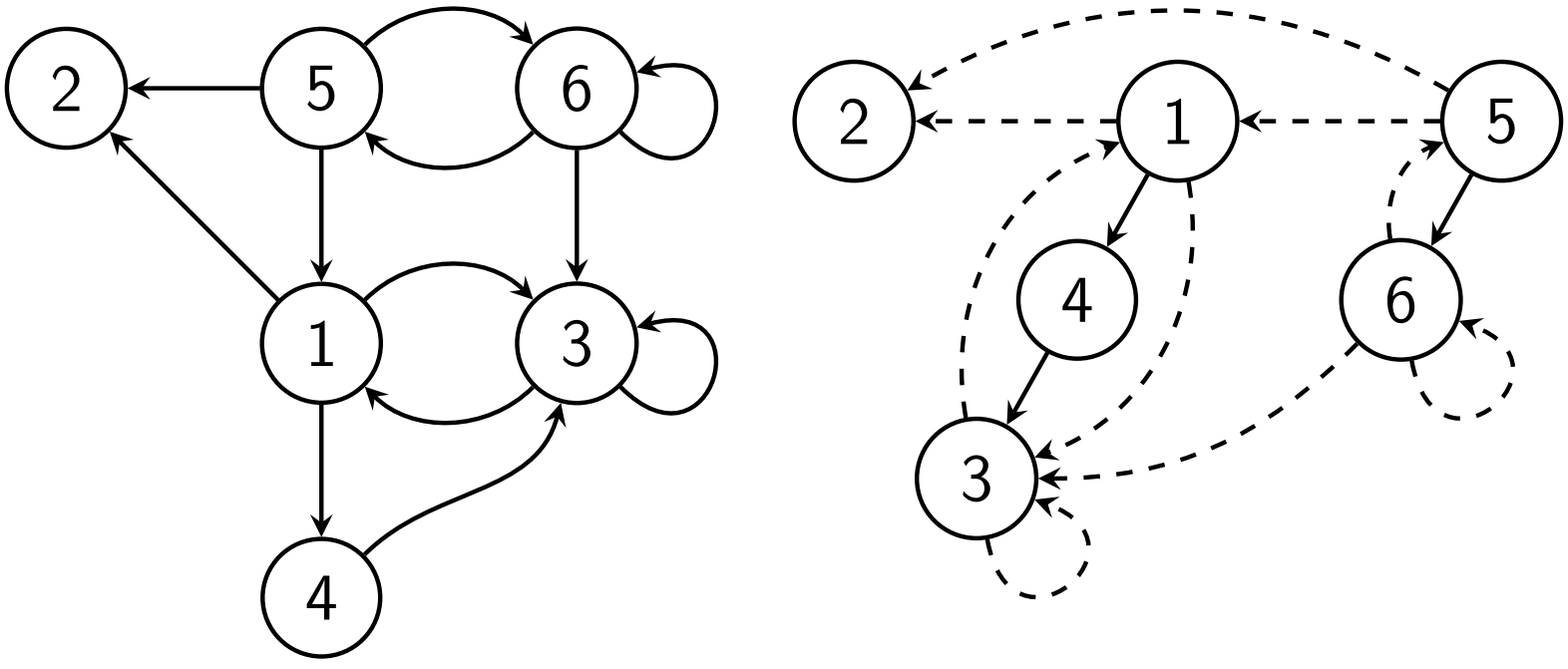
Idag

- Starkt sammanhängande komponenter.
- Duggaresultat.
- Sökträd.

Starkt sammanhängande komponenter

Uppspännande skog

Graf, och en möjlig skog:



I Haskell

```
type Forest = [Tree]
data Tree   = Root Vertex [Edge]
type Vertex = Integer
data Edge   = Regular Tree
            | Other Kind Vertex
data Kind   = Forward | Back | Cross

-- Börjar varje sökning i den första obesökta
-- listnoden.
dfs :: Graph -> [Vertex] -> Forest

dff :: Graph -> Forest
dff g = dfs g (vertices g)
```

Traversering

```
-- Alla trädnoder, i preorder (vänster till höger).
preorderLR :: Tree -> [Vertex]

-- Alla trädnoder, i preorder (höger till vänster).
-- Motsvarar att traversera skogen i postordning
-- (vänster till höger), och pusha noderna på en
-- till att börja med tom stack.
preorderRL :: Forest -> [Vertex]
```

SCC-algoritm

```
-- Reverserar grafen.
opposite :: Graph -> Graph

-- Resultatet är omvänt topologiskt sorterat.
sccs :: Graph -> [[Vertex]]
sccs g = map preorderLR (dfs g stack)
  where
  stack = preorderRL (dff (opposite g))
```

Ger följande funktion en lista av starkt sammanhängande komponenter som svar?

```
sccs :: Graph -> [[Vertex]]
sccs g = map preorderLR (dfs (opposite g) stack)
  where
  stack = preorderRL (dff g)
```

Dugga

Duggaresultat

Uppgift 1 42/73.
Uppgift 2 24/73.
Uppgift 3 16/73.

Uppgift 1

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        m.insert(i + j, q.delete-min());
    }
}
```

- `insert`: $O(1)$.
  Totalt: $O(\mathtt{n}^2)$.
- `delete-min`: $O(\log |\mathtt{q}|)$.
  Totalt: $O(\mathtt{n}^2 \log \mathtt{n}^2) = O(\mathtt{n}^2 \log \mathtt{n})$.
- Totalt: $O(\mathtt{n}^2 + \mathtt{n}^2 \log \mathtt{n}) = O(\mathtt{n}^2 \log \mathtt{n})$.
- Noggrannare analys: $\Theta(\mathtt{n}^2 \log \mathtt{n})$.

Uppgift 1

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        m.insert(i + j, q.delete-min());
    }
}
```

Vanliga misstag:

- $O(\log(n^2)) = O(n)$, totalt $O(n^3)$.
  Däremot:
  - $O(\log(2^n)) = O(n)$.
  - $O(\log(n^2)) = O(\log n)$.
- $O(n^2 \cdot n) = O(n^2)$ (?).
  Däremot: $O(n^2 + n) = O(n^2)$.

Uppgift 2

Implementera en metod som sätter in ett element *först* i en cirkulär array, på konstant tid:

```
A[] queue; // Arrayen. Invariant: queue.length > 0.
int size;  // Köns storlek.
           // Invariant: 0 <= size <= queue.length.
int front; // Invariant: 0 <= front <= queue.length.
           // Om size > 0 pekar front på köns första
           // element. Om size > 1 pekar
           // (front + 1) % queue.length på köns
           // andra element. Och så vidare.
```

```
// Sätter in a först i kön. Om kön är full lämnas den
// oförändrad. Resultatet är true om elementet sattes
// in, och annars false. Tidskomplexitet: O(1).
public boolean push(A a) {
    if (size == queue.length) {
        return false;
    }

    size++;
    if (front <= 0) {
        front = queue.length;
    }
    front--;
    queue[front] = a;

    return true;
}
```

Uppgift 2

Några misstag:

- Inte *cirkulär* array.
- Inte konstant tid (flyttar alla element).
- Användning av länkade listor. (?)
- Insättning *sist* i kön.
- Ej `size++`.

Uppgift 2

Några misstag:

- Inte *cirkulär* array.
- Inte konstant tid (flyttar alla element).
- Användning av länkade listor. (?)
- Insättning *sist* i kön.
- Ej `size++`. Testa!

Uppgift 3

- `new Priority-queue(`P`)`/`empty(`P`)`: $O(1)$.
- `insert(`v`,`p`)`: $O(\log n)$.
- `delete-min()`: $O(\log n)$.
- `delete-small()`: Tar bort alla värden vars prioritet är mindre än P, $O(1)$.

Uppgift 3

Två prioritetsköer:

- En för värden vars prioritet är mindre än P.
- En för värden med högre prioritet.

Uppgift 3

Några problem:

- Det är svårt att implementera en heap från grunden. Använd standarddatastrukturer!
- Hashtabell istället för prioritetskö.
- `delete-small`: Ej $O(1)$.

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 *total ordning*.

Komparator i Haskell

Typklassen `Ord` innehåller `compare`:

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

data Ordering = LT | EQ | GT

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)`.
- `find-min()`.
- `delete-min()`.
- `delete(k)`.
- `iterator()`:
  Går igenom elementen i sorterad ordning.

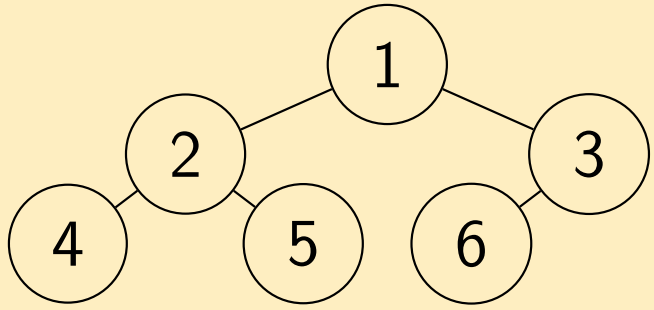
Binära sökträd

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

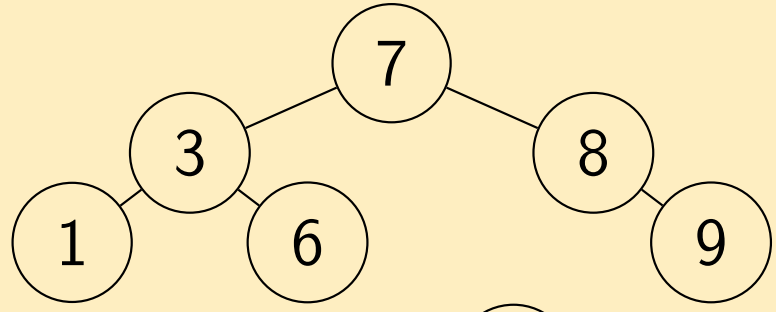
- `empty`: Tom mängd/avbildning.
- `member k t`/`lookup k t`.
- `insert k t`/`insert k v t`.
- `delete-min t`.
- `delete k t`.
- `inorder t`:
  En sorterad lista med alla element.

Vilka träd är binära sökträd?

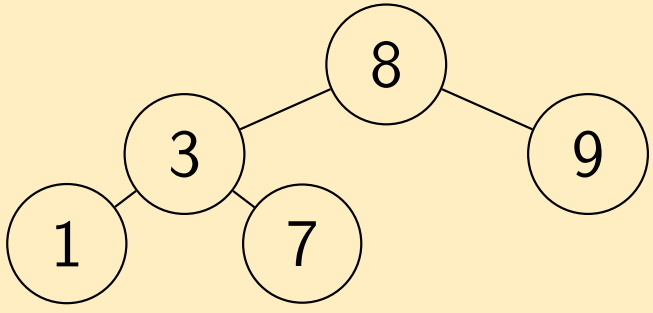
A:



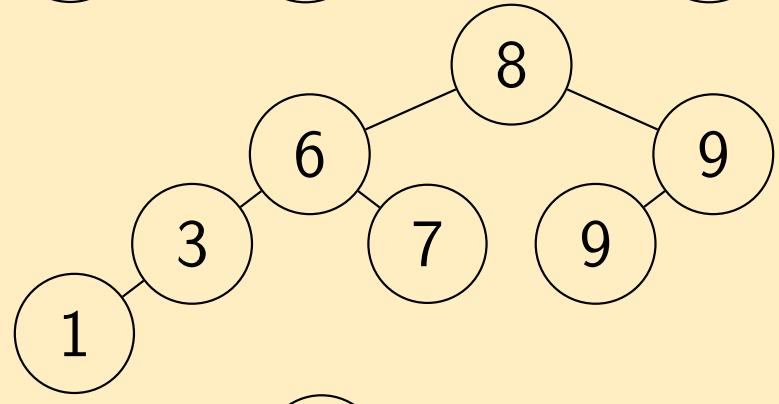
B:



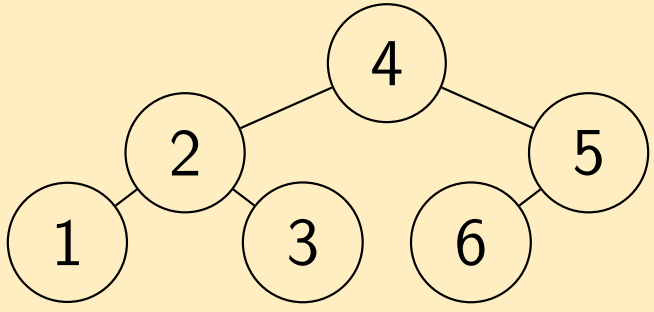
C:



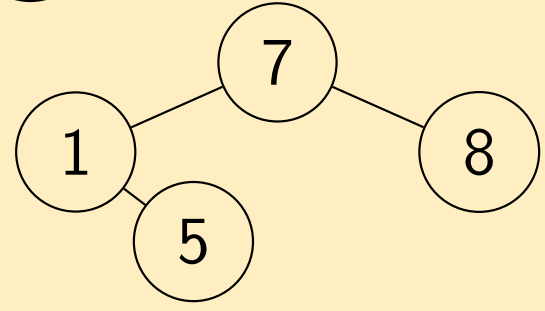
D:



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)

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

```
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)

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

```
inorder :: Tree a -> [a]
inorder Empty          = []
inorder (Node l x r) = inorder l ++ (x : inorder r)
```

Obalanserade binära sökträd

```
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:

```
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)
```

Obalanserade binära sökträd

Kanske mer idiomatiskt:

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

Obalanserade binära sökträd

En möjlig implementation:

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

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

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

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

Obalanserade binära sökträd

Iterativ kod:

```
public boolean member(A a) {
    TreeNode 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 TreeNode insert(A a, TreeNode n) {
    if (n == null) return new TreeNode(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

Värstafallstidskomplexitet:

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

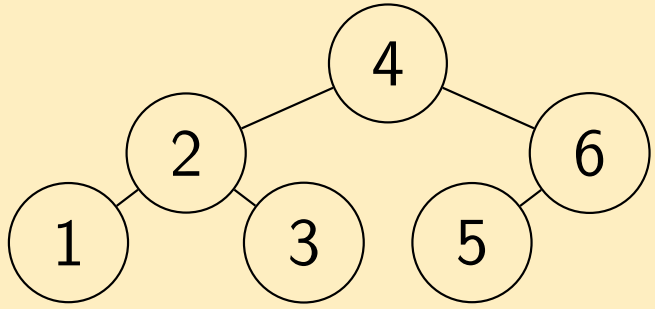
- AVL-träd (Adelson-Velskii & Landis).
- Röd-svarta träd.
- B^+-träd.
- ...

AVL-träd

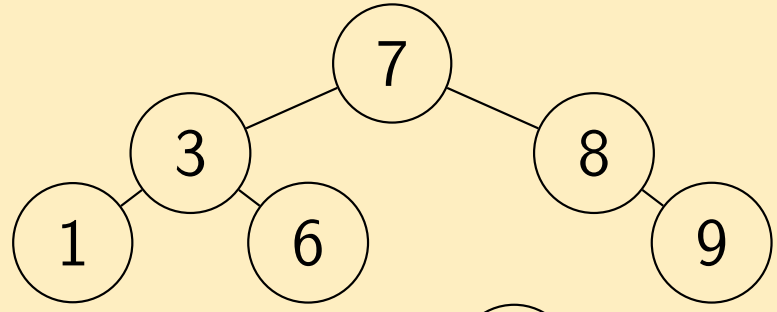
- Invariant (för varje nod):
  Vänster och höger delträd har samma höjd, ± 1.
- Operationer som ändrar trädets struktur
  använder (ibland) *rotationer* för att
  återställa invarianten.

Vilka träd är AVL-träd?

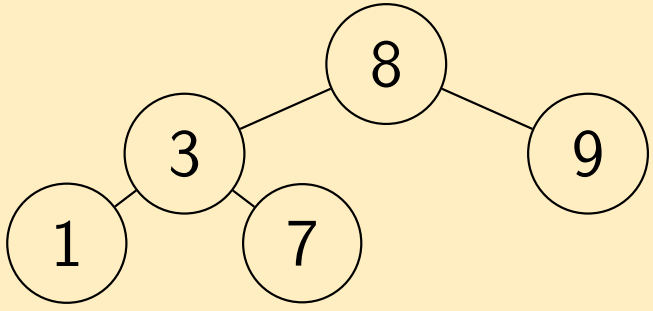
A:



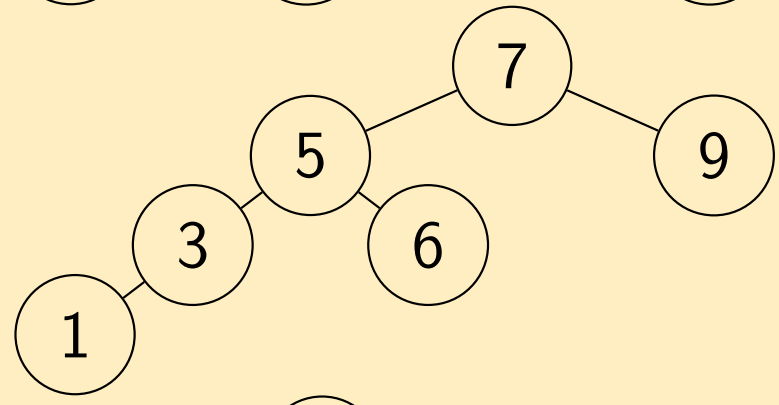
B:



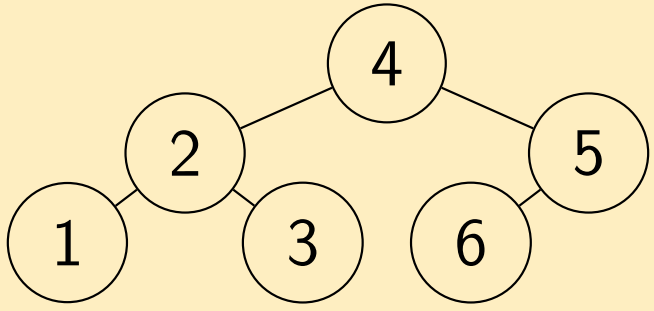
C:



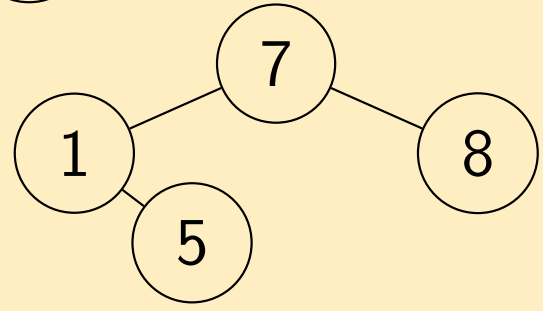
D:



E:



F:



Insättning i AVL-träd

Skiss av en algoritm:

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

Sammanfattning

Idag:

- Starkt sammanhängande komponenter.
- Sökträd.

Nästa gång: Mer om sökträd.