Föreläsning
Datastrukturer (DAT037)

Nils Anders Danielsson

2015-11-09

Repetition

Förra gången:

- Amorterad tidskomplexitet.
- Listor, stackar, köer.
- Länkade listor, pekarjonglering.
- Invarianter, assertions m m, testning.

Idag

- Mer om listor.
- Persistenta datastrukturer.
- Cirkulära arrayer.
- Köer m h a två listor.
- Binära träd.
- Prioritetsköer.

Listor i Haskell

Haskell

- Enkellänkade listor.
- Två slags noder:
  tom lista (`[]`) och conscell (`:`).
- Endast pekare till första noden,
  inget objekt innehållandes storlek e d.
- Inga vaktposter.
- Exempel:

```
xs = [2, 3] = 2 : 3 : []
```

Iteratorer

Listor används ofta som iteratorer:

- Istället för `hasNext`/`next` används mönstermatchning.
- `remove` finns inte.

Persistens

- (Vanliga) Haskellistor kan inte uppdateras.
- När man ”ändrar” en lista
  så finns den gamla listan kvar
  (tills den skräpsamlas).
- Svansar kan delas.

Persistens

Exempel:

```
xs = [2, 3]
ys = 1 : xs
```

Här delas `[2, 3]` mellan två listor.

Persistens

Append:

```
(++) :: [a] -> [a] -> [a]
[]       ++ bs = bs
(a : as) ++ bs = a : (as ++ bs)
```

Exempel:

```
xs = [2, 3]
zs = xs ++ [4]
```

Här delas bara listelementen (2 och 3).

Persistens

- Fördel: Kanske enklare att förstå, undvika buggar.
- Nackdel: Vissa operationer mindre effektiva.
- Kan implementera persistenta datastrukturer i Java också.

Hur många consceller innehåller
`tails [1, 2, 3]`
(när uttrycket är fullt evaluerat)?

```
tails :: [a] -> [[a]]
tails [] = [] : []
tails xs = xs : tails (tail xs)

tails [1, 2, 3] =
  [[1, 2, 3], [2, 3], [3], []]
```

(`tail (a : as) = as`.)

Listor, sammanfattning

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista	Haskellista
`add(x)`			
`add(x, i)`			
`remove(x)`			
`remove(i)`			
`get(i)`			
`set(i, x)`			
`contains(x)`			
`size`			
`iterator`			
`hasNext/next`			
`remove`			

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista	Haskellista
`add(x)`	$O(1)$ am	$O(1)$	$O(1)$
`add(x, i)`			
`remove(x)`			
`remove(i)`			
`get(i)`			
`set(i, x)`			
`contains(x)`			
`size`			
`iterator`			
`hasNext/next`			
`remove`			

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista	Haskellista
`add(x)`	$O(1)$ am	$O(1)$	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$	$O(n)$
`remove(x)`			
`remove(i)`			
`get(i)`			
`set(i, x)`			
`contains(x)`			
`size`			
`iterator`			
`hasNext/next`			
`remove`			

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista	Haskellista
`add(x)`	$O(1)$ am	$O(1)$	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$	$O(n)$
`remove(i)`			
`get(i)`			
`set(i, x)`			
`contains(x)`			
`size`			
`iterator`			
`hasNext/next`			
`remove`			

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista	Haskellista
`add(x)`	$O(1)$ am	$O(1)$	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$	$O(n)$
`get(i)`			
`set(i, x)`			
`contains(x)`			
`size`			
`iterator`			
`hasNext/next`			
`remove`			

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista	Haskellista
`add(x)`	$O(1)$ am	$O(1)$	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$	$O(n)$
`get(i)`	$O(1)$	$O(n)$	$O(n)$
`set(i, x)`			
`contains(x)`			
`size`			
`iterator`			
`hasNext/next`			
`remove`			

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista	Haskellista
`add(x)`	$O(1)$ am	$O(1)$	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$	$O(n)$
`get(i)`	$O(1)$	$O(n)$	$O(n)$
`set(i, x)`	$O(1)$	$O(n)$	$O(n)$
`contains(x)`			
`size`			
`iterator`			
`hasNext/next`			
`remove`			

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista	Haskellista
`add(x)`	$O(1)$ am	$O(1)$	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$	$O(n)$
`get(i)`	$O(1)$	$O(n)$	$O(n)$
`set(i, x)`	$O(1)$	$O(n)$	$O(n)$
`contains(x)`	$O(n)$	$O(n)$	$O(n)$
`size`			
`iterator`			
`hasNext/next`			
`remove`			

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista	Haskellista
`add(x)`	$O(1)$ am	$O(1)$	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$	$O(n)$
`get(i)`	$O(1)$	$O(n)$	$O(n)$
`set(i, x)`	$O(1)$	$O(n)$	$O(n)$
`contains(x)`	$O(n)$	$O(n)$	$O(n)$
`size`	$O(1)$	$O(1)$	$O(n)$
`iterator`			
`hasNext/next`			
`remove`			

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista	Haskellista
`add(x)`	$O(1)$ am	$O(1)$	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$	$O(n)$
`get(i)`	$O(1)$	$O(n)$	$O(n)$
`set(i, x)`	$O(1)$	$O(n)$	$O(n)$
`contains(x)`	$O(n)$	$O(n)$	$O(n)$
`size`	$O(1)$	$O(1)$	$O(n)$
`iterator`	$O(1)$	$O(1)$	
`hasNext/next`			
`remove`			

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista	Haskellista
`add(x)`	$O(1)$ am	$O(1)$	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$	$O(n)$
`get(i)`	$O(1)$	$O(n)$	$O(n)$
`set(i, x)`	$O(1)$	$O(n)$	$O(n)$
`contains(x)`	$O(n)$	$O(n)$	$O(n)$
`size`	$O(1)$	$O(1)$	$O(n)$
`iterator`	$O(1)$	$O(1)$	
`hasNext/next`	$O(1)$	$O(1)$	$O(1)$
`remove`			

Listor: tidskomplexitet

	Dynamisk array	Dubbellänkad lista	Haskellista
`add(x)`	$O(1)$ am	$O(1)$	$O(1)$
`add(x, i)`	$O(n)$	$O(n)$	$O(n)$
`remove(x)`	$O(n)$	$O(n)$	$O(n)$
`remove(i)`	$O(n)$	$O(n)$	$O(n)$
`get(i)`	$O(1)$	$O(n)$	$O(n)$
`set(i, x)`	$O(1)$	$O(n)$	$O(n)$
`contains(x)`	$O(n)$	$O(n)$	$O(n)$
`size`	$O(1)$	$O(1)$	$O(n)$
`iterator`	$O(1)$	$O(1)$	
`hasNext/next`	$O(1)$	$O(1)$	$O(1)$
`remove`	$O(n)$	$O(1)$	

Listor: tidskomplexitet

Några kommentarer:

- `remove`, `contains`:
  Givet att jämförelser tar konstant tid.
- Indexbaserade operationer för länkade listor:
  Bättre än $\Theta(n)$ om man vet att `i` pekar
  ”nära” början (eller i vissa fall slutet) av listan.

Cirkulära arrayer

Cirkulära arrayer

Implementationsteknik för köer.

Cirkulära arrayer

```
public class CircularArrayQueue<A> {
    private A[] queue;
    private int size;
    private int front; // Nästa element (eller rear).
    private int rear;  // Nästa lediga (eller front).

    // Precondition: capacity >= 0.
    public CircularArrayQueue(int capacity) {
        if (capacity < 0) {
            throw new NonPositiveCapacityException();
        }
        queue = (A[]) new Object[capacity];
        size = front = rear = 0;
    }

    ...
```

Cirkulära arrayer

```
// Precondition: Kön får inte vara full.
public void enqueue(A a) {
    if (size == queue.length) {
        throw new FullQueueException();
    }

    size++;
    queue[rear] = a;
    rear = (rear + 1) % queue.length;
}
```

Cirkulära arrayer

```
// Precondition: Kön får inte vara tom.
public A dequeue() {
    if (size == 0) {
        throw new EmptyQueueException();
    }

    size--;
    A a = queue[front];
    queue[front] = null;  // Undvik minnesläckor.
    front = (front + 1) % queue.length;

    return a;
}
```

Vilka datastrukturer ger $O(1)$ `enqueue` och `dequeue` (ev amorterat) för FIFO-köer?

- Dynamiska arrayer (linjära).
- Cirkulära, dynamiska arrayer.
- Enkellänkade listor utan pekare till sista noden.
- Enkellänkade listor med pekare till sista noden.
- Dubbellänkade listor med pekare till två vaktposter (först och sist).

Köer i Haskell

Köer i Haskell

- Implementera kö-ADTn m h a lista?
- Nja, dyrt att sätta in element sist.
- Ett alternativ: två listor.

Köer i Haskell

```
module Queue
  (Queue, empty, isEmpty, enqueue, first, dequeue)
  where

-- Invariant: front är tom omm kön är tom.
data Queue a = Q { front :: [a], rear :: [a] }

empty :: Queue a
empty = Q [] []

isEmpty :: Queue a -> Bool
isEmpty (Q [] _) = True
isEmpty _        = False
```

Köer i Haskell

```
-- Invariant: front är tom omm kön är tom.
data Queue a = Q { front :: [a], rear :: [a] }

enqueue :: a -> Queue a -> Queue a
enqueue x (Q [] _) = Q [x] []
enqueue x (Q f r)  = Q f (x : r)

first :: Queue a -> Maybe a
first (Q []      _) = Nothing
first (Q (x : _) _) = Just x

dequeue :: Queue a -> Maybe (Queue a)
dequeue (Q []      _) = Nothing
dequeue (Q [_]     r) = Just (Q (reverse r) [])
dequeue (Q (_ : f) r) = Just (Q f r)
```

Köer i Haskell

Tidskomplexitet (jag ignorerar lathet):

- `empty`, `isEmpty`, `enqueue`, `first`: $\Theta(1)$.
- `dequeue`: $O(1)$ (amorterat).
  Betala ett mynt extra vid insättning,
  använd mynten för att betala för `reverse`.

Skapa en kö innehållandes $1, 2, ..., n$:

```
xs = enqueue n (… (enqueue 2 (enqueue 1 empty)) …)
```

Kör sedan följande kod n ggr:

```
dequeue xs
```

Vad är tidskomplexiteten?

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

Persistens, igen

- Analysen fungerar inte eftersom vi betalar med samma mynt många gånger.
- Den amorterade tidskomplexiteten för `dequeue` är $O(1)$ om kön används *enkeltrådat*.
- Man kan konstruera persistenta köer med bra "flertrådad" tidskomplexitet, se t ex Okasakis avhandling.

Binära träd

Binära träd

- Ett binärt träd är tomt eller
  en nod (ev med ett värde)
  plus två delträd, till vänster och till höger.
- Terminologi:
  - Förälder, barn, syskon, barnbarn o s v.
  - Rot, löv.

Binära träd

En representation:

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

Exempel:

```
tree :: Tree Integer
tree = Node (Node Empty 2 Empty)
            1
            (Node Empty 3 (Node Empty 5 Empty))
```

Binära träd

En annan representation:

```
class Tree<A> {
  class TreeNode {
    A        contents;
    TreeNode left;   // null if left child is missing.
    TreeNode right;  // null if right child is missing.
  }

  TreeNode root;  // null if tree is empty.
}
```

Binära träd

Höjd:

- Tomma träd har höjd -1.
- Annars:
  Antalet steg från roten till djupaste lövet.

Binära träd

Höjd:

- Tomma träd har höjd -1.
- Annars:
  Antalet steg från roten till djupaste lövet.

```
height :: Tree a -> Integer
height Empty        = -1
height (Node l _ r) = 1 + max (height l) (height r)
```

Binära träd

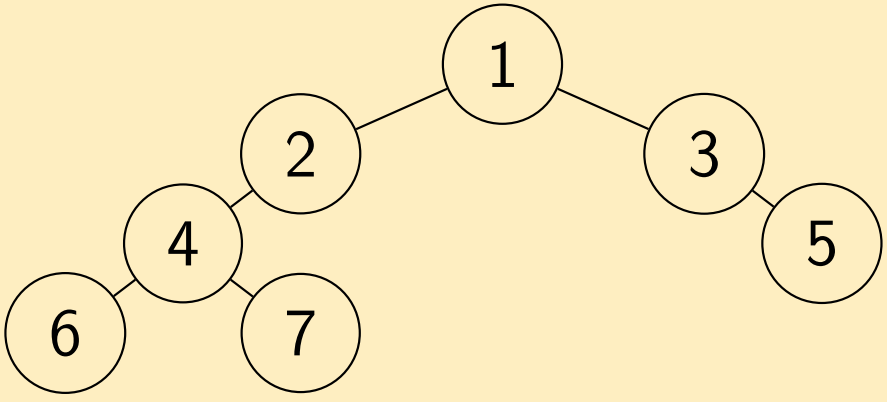
Höjd:

- Tomma träd har höjd -1.
- Annars:
  Antalet steg från roten till djupaste lövet.

```
int height(TreeNode n) {
  if (n == null) {
    return -1;
  } else {
    return 1 + Math.max(height(n.left),
                        height(n.right));
  }
}
```

```
int s(TreeNode n) {
  if (n == null) {
    return 0;
  } else {
    return 1 + s(n.left) + s(n.right);
  }
}
```

Vad blir resultatet av att applicera s på följande träds rot?



Prioritetsköer

Prioritetsköer

Köer där varje element har viss prioritet.

Gränssnitt (exempel):

- Konstruerare för tom kö.
- `insert`: Lägger till element.
- `find-min`: Ger tillbaka minsta elementet.
- `delete-min`:
  Tar bort och ger tillbaka minsta elementet.
- `increase-key`/`decrease-key`/`modify-key`:
  Ändrar ett elements prioritet.
- `merge`: Slår ihop två köer.

Prioritetsköer

Några tillämpningar:

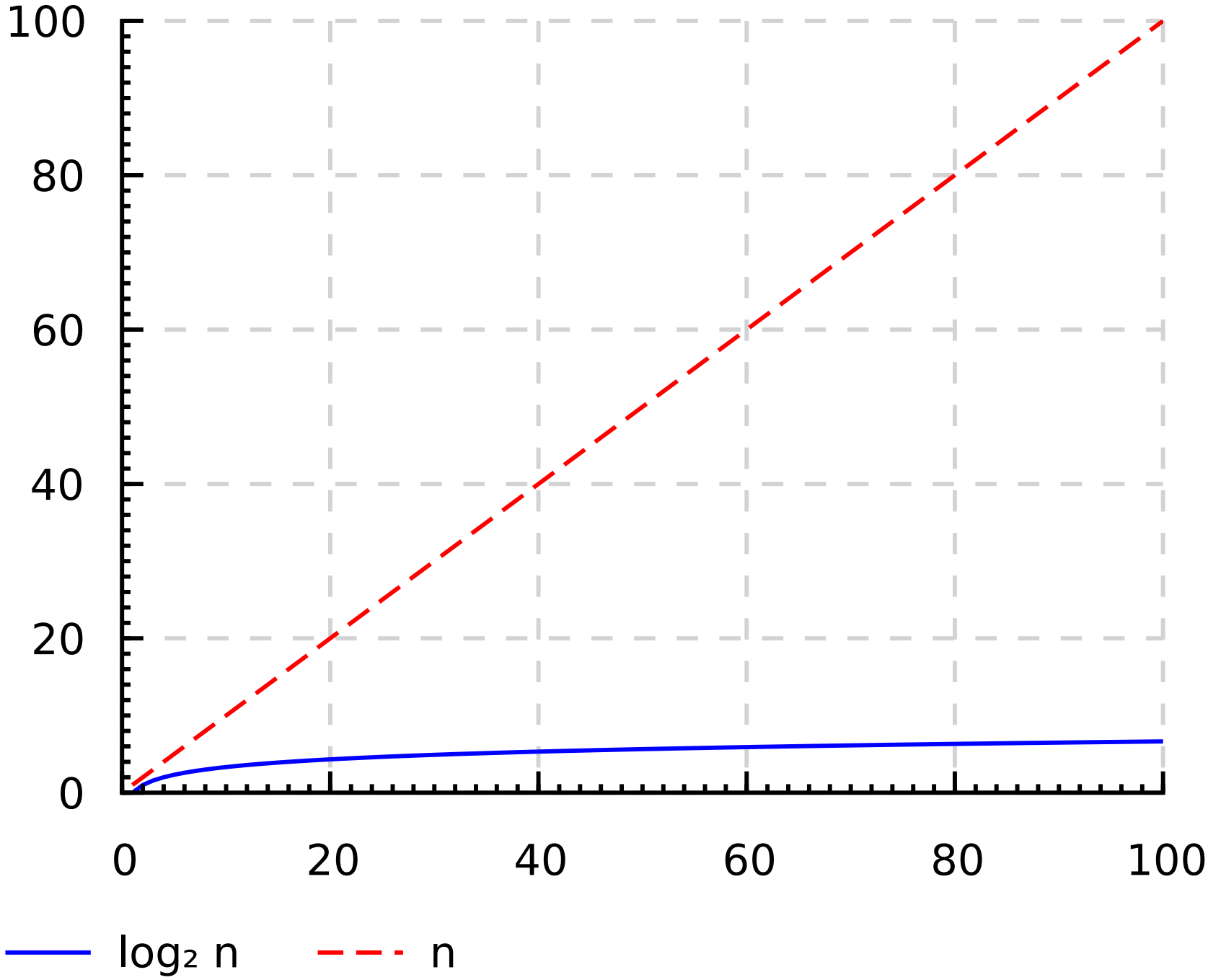
- Schemaläggning av processer.
- Sortering.
- Dijkstras algoritm (labb 3).

Labb 2: Implementera prioritetskö.

Om man implementerar prioritetskö-ADTn med listor, vad blir tidskomplexiteten (ev amorterad) för `insert` och `delete-min`?

- `insert`: $\Theta(1)$, `delete-min`: $\Theta(1)$.
- `insert`: $\Theta(1)$, `delete-min`: $\Theta(n)$.
- `insert`: $\Theta(n)$, `delete-min`: $\Theta(1)$.
- `insert`: $\Theta(n)$, `delete-min`: $\Theta(n)$.

Anta att prioriteter kan jämföras på konstant tid.


100
80
60
40
20
0
0
20
40
60
80
100
log₂ n
n

Binära heapar

Binära heapar

Kompletta binära träd med heapordningsegenskapen.

Heapordningsegenskapen

Varje nod är mindre än eller lika med alla sina barn.

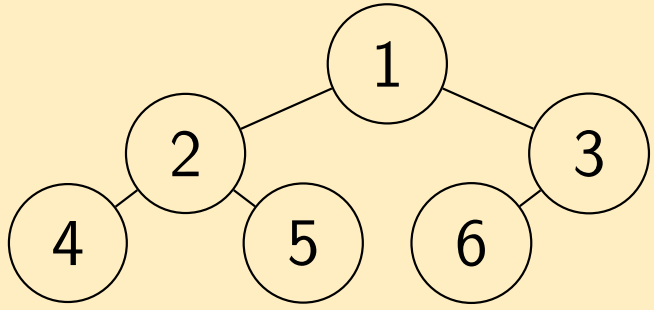
Komplett binärt träd

Så lågt som möjligt, alla nivåer helt fyllda utom möjligtvis den sista, som är fylld från vänster.

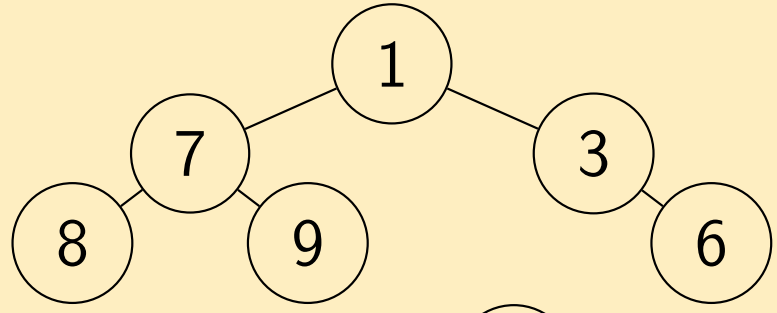
En binär heap med n noder har höjden $\Theta(\log n)$.

Identifiera alla binära heapar.

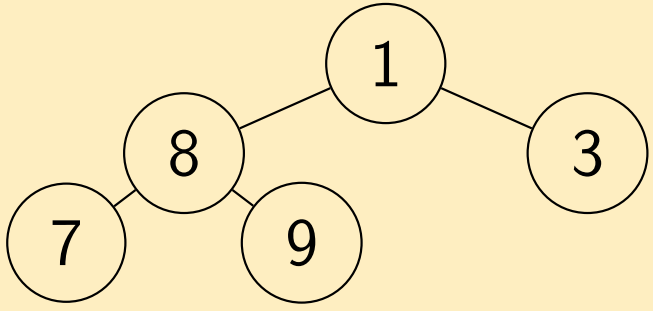
A:



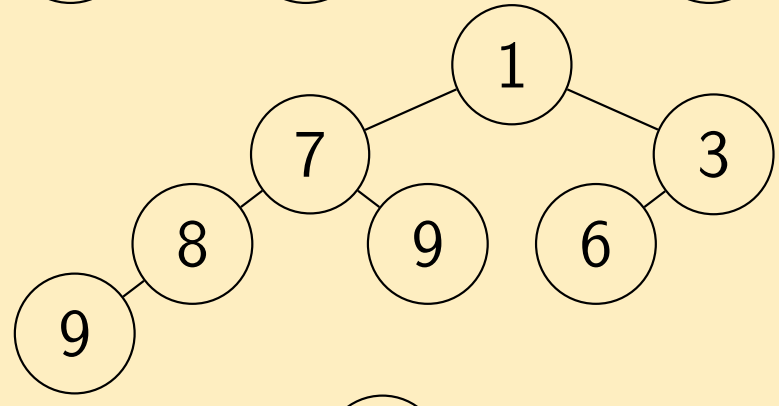
B:



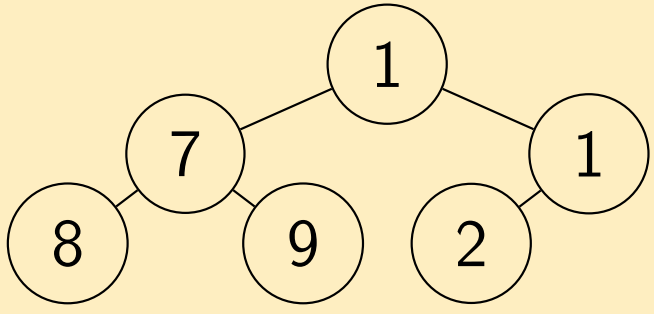
C:



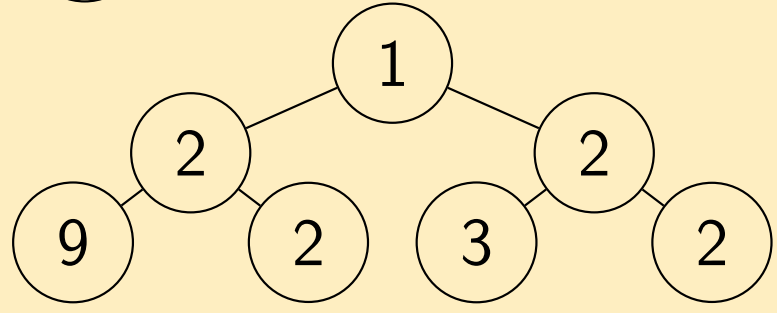
D:



E:



F:



Sammanfattning

- Cirkulära arrayer.
- Köer m h a två listor.
- Binära träd.
- Prioritetsköer.

Nästa gång:

- Mer om prioritetsköer.