Föreläsning
Datastrukturer (DAT036)

Nils Anders Danielsson

2013-11-06

Idag

Prioritetsköer:

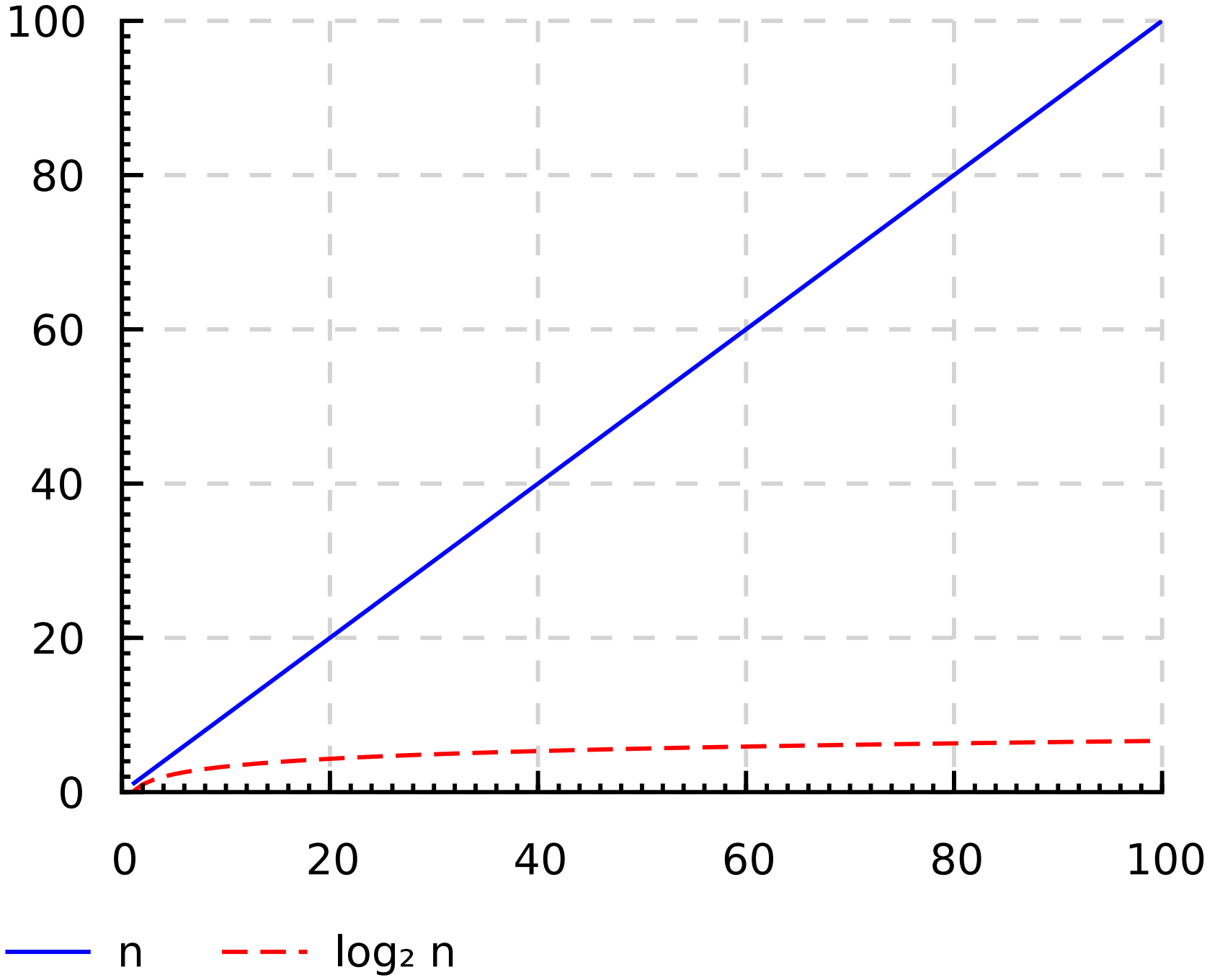
- Binära heapar.
- Braunheapar.
- Binomialheapar.

Prioritetsköer: repetition

Köer där varje element har viss prioritet.

Gränssnitt (exempel):

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


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

Binära heapar

Kompletta binära träd med heapordningsegenskapen.

Heapordningsegenskapen

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

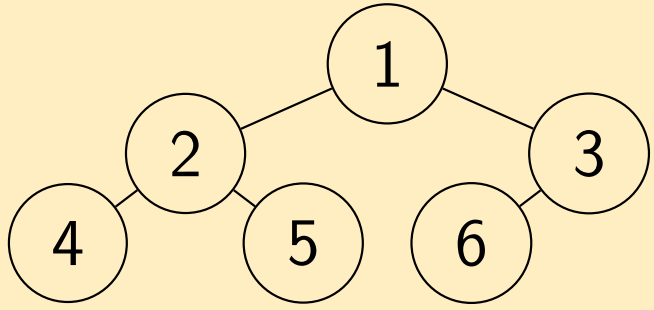
Komplett binärt träd

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

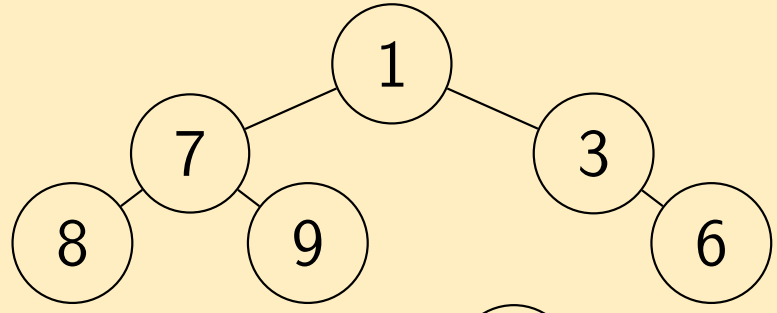
En binär heap med n noder har höjden $\Theta(\log n)$.

Vilka träd är binära heapar?

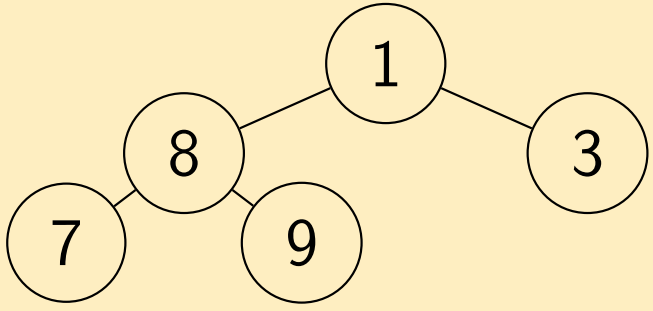
A:



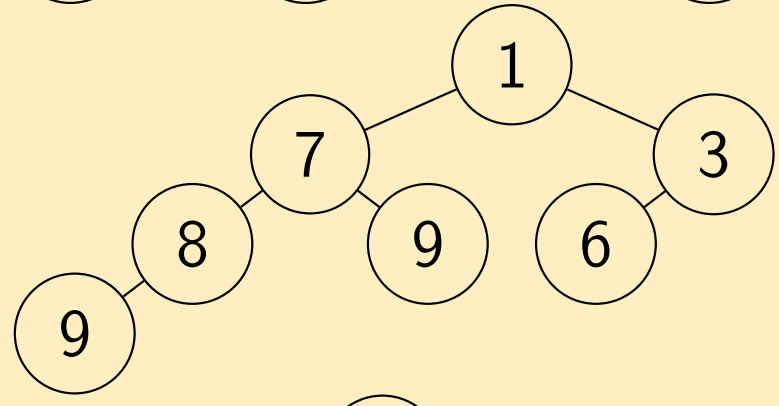
B:



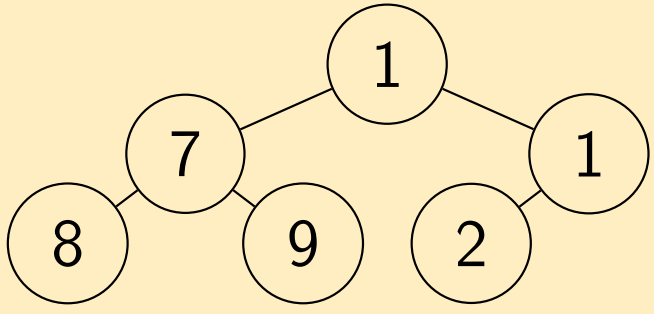
C:



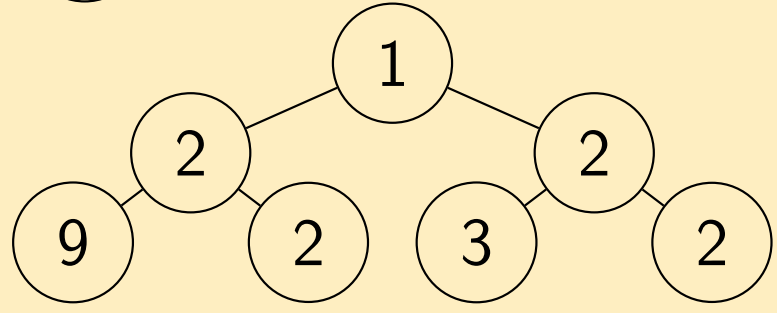
D:



E:



F:



Implementation av binära heapar

- Tom kö: Tomt träd.
- `find-min`: Ge tillbaka roten.
- `insert`: Stoppa in sist. Bubbla upp.
- `delete-min`: Ta bort roten.
  Stoppa in sista elementet överst. Bubbla ned.
  Ge tillbaka gamla roten.
- `decrease-key`: Ändra prioritet,
  bubbla åt rätt håll.

Bubbla upp/ned tills heapordningsegenskapen återställts.

Tidskomplexitet

Om man kan hitta noderna snabbt:

- Tom kö: $\Theta(1)$.
- `find-min`: $\Theta(1)$.
- `insert`: $O(\log n)$ (kanske amorterat).
- `delete-min`: $O(\log n)$ (kanske amorterat).
- `decrease-key`: $O(\log n)$.

(Givet att jämförelser tar konstant tid.)

De flesta noderna är långt ned i trädet.
Medelkomplexitet för `insert`: $O(1)$.

Implementation av binära heapar

Kan representera trädet med array (labb 2).

- Roten på position 1.
- Sista elementet på position n.
- Första tomma cellen på position $n + 1$.
- Nod is vänstra barn: $2i$.
- Nod is högra barn: $2i + 1$.
- Nod is förälder ($i > 1$): $\lfloor i/2 \rfloor$.

`decrease-key`

När man implementerar `decrease-key`,
hur hittar man noden snabbt?

Kan använda extra datastruktur.

Exempel: Hashtabell.

`build-heap`

`build-heap`: Konverterar lista e d till heap.

- Stoppa in alla elementen i godtycklig ordning.
- Bubbla *ned* ett element i taget,
  med början på det nedersta, högraste.
  (Kan skippa alla löv.)

Heapordningsegenskapen uppfylls
(kan bevisas med induktion).

`build-heap`: tidskomplexitet

- Tid för viss nod: $O(\text{nodens höjd})$.
  (Givet $O(1)$-jämförelser.)
- Total tid: $O(\text{summan av alla höjder})$.
- Nästan alla noder är långt ned.
- Total tid: $\Theta(n)$.
- Bevis: Se boken.

Heapsort

Konstruera en sorteringsalgoritm baserad på binära heapar. Vad är dess värstafallstidskomplexitet, givet att jämförelser tar konstant tid?

- $O(1)$.
- $O(\log n)$.
- $O(n)$.
- $O(n \log n)$.
- $O(n^2)$.
- $O(n^2 \log n)$.

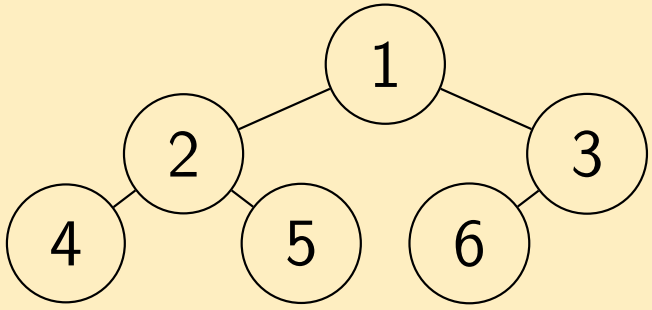
Persistent prioritetskö

Haskell:

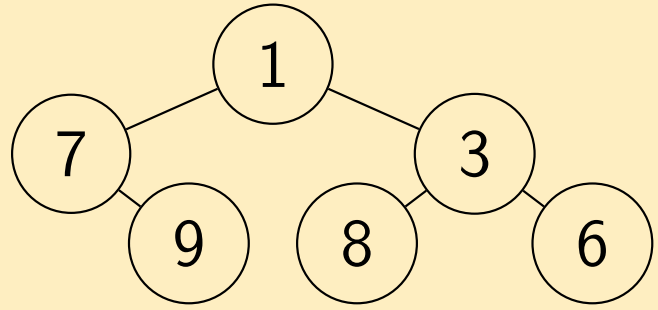
- Uppdatering av ett (eller alla) element i en vanlig persistent array av storlek n har tidskomplexiteten $\Theta(n)$.
- Man kan använda vanliga, effektivt uppdaterbara arrayer, men de är inte persistenta.
- Alternativ: Heapordnade Braunträd (labb 2).

Vilka träd är Braunheapar?

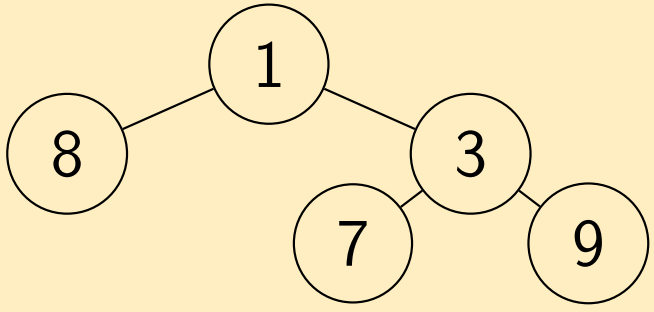
A:



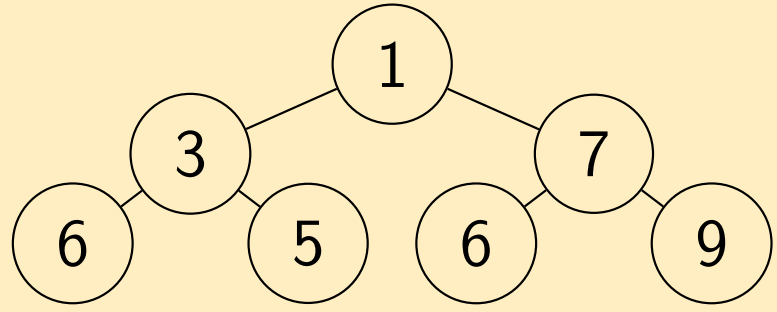
B:



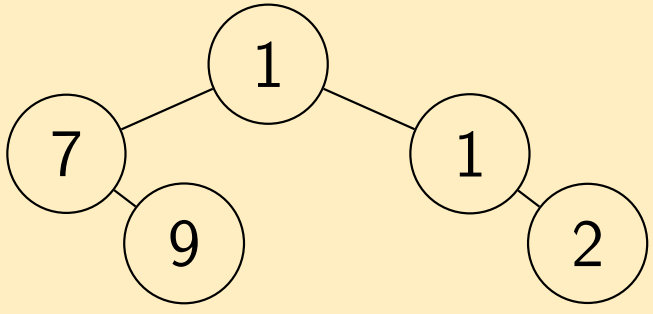
C:



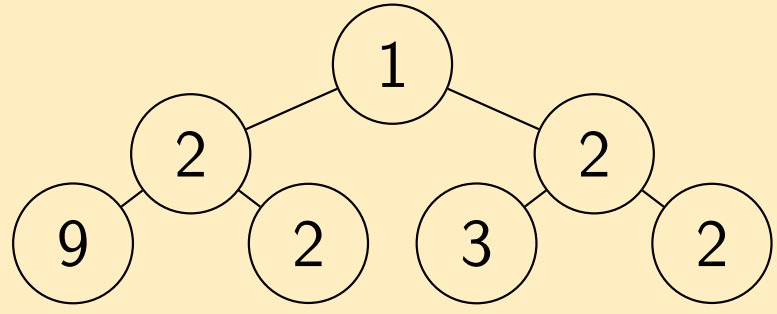
D:



E:



F:



Persistent prioritetskö

```
module BraunHeap
  ( PriorityQueue
  , empty, isEmpty
  , insert
  , findMin, deleteMin
  )
  where

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

-- Invarianter för Node x l r:
-- * 0 <= size r - size l <= 1.
-- * x är mindre än eller lika med
--   alla element i l och r.
```

Persistent prioritetskö

```
empty :: PriorityQueue a
empty = Empty

isEmpty :: PriorityQueue a -> Bool
isEmpty Empty          = True
isEmpty (Node _ _ _) = False
```

Persistent prioritetskö

```
-- Invarianter för Node x l r:
-- * 0 <= size r - size l <= 1.
-- * x är mindre än eller lika med
--   alla element i l och r.

insert :: Ord a =>
          a -> PriorityQueue a -> PriorityQueue a
insert x Empty          = Node x Empty Empty
insert x (Node y l r)
  | x <= y    = Node x r (insert y l)
  | otherwise = Node y r (insert x l)
```

Persistent prioritetskö

```
-- Precondition: Kön får inte vara tom.
findMin :: PriorityQueue a -> a
findMin Empty          = error "findMin: Empty queue."
findMin (Node x _ _) = x

-- Precondition: Kön får inte vara tom.
deleteMin :: Ord a =>
             PriorityQueue a -> (a, PriorityQueue a)
deleteMin Empty                = error "deleteMin: Empty."
deleteMin (Node x _ Empty) = (x, Empty)
deleteMin (Node x l r)     = (x, bubbleDown y r' l)
  where (y, r') = removeRightMost r
```

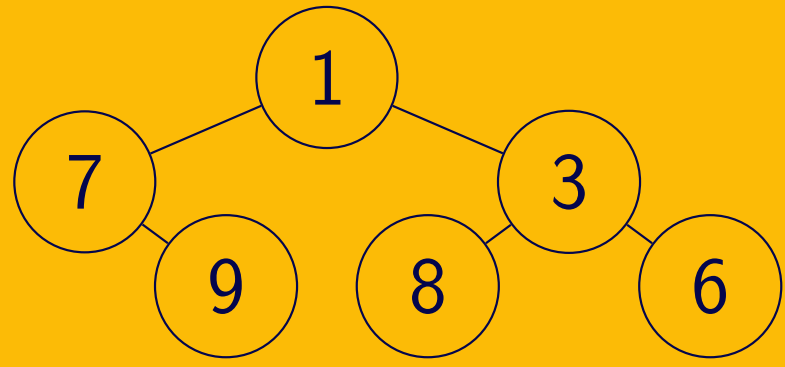
Persistent prioritetskö

```
-- Precondition: Kön får inte vara tom.
removeRightMost ::
  PriorityQueue a -> (a, PriorityQueue a)

-- Precondition för bubbleDown x l r:
-- 0 <= size r - size l <= 1.
bubbleDown ::
  Ord a =>
  a -> PriorityQueue a -> PriorityQueue a ->
  PriorityQueue a
```

Om man kör `deleteMin q`,

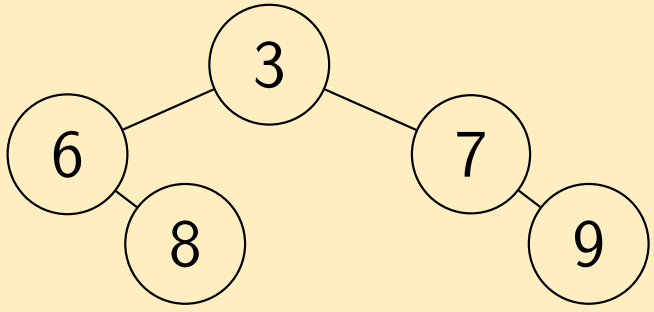
där q =



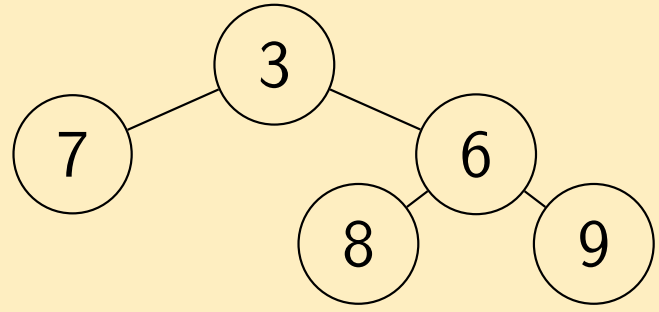
,

hur ser det nya trädet ut?

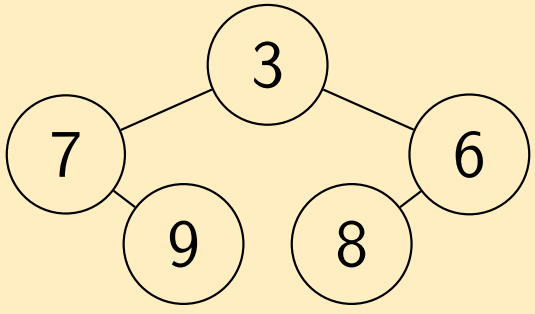
A:



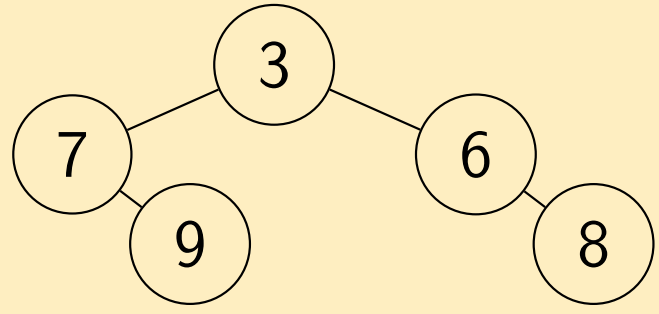
B:



C:



D:



Tidskomplexitet

- `empty`: $\Theta(1)$.
- `isEmpty`: $\Theta(1)$.
- `findMin`: $\Theta(1)$.
- `insert`: $\Theta(\log n)$.
- `deleteMin`: $\Theta(\log n)$.
- `buildHeap`: $\Theta(n)$.
- `decreaseKey`: $\Theta(n)$.

(Givet att jämförelser tar konstant tid.)

merge

- $\Theta(n)$ om man använder binära heapar implementerade med arrayer.
- Med binomialheapar (labb 2): $O(\log n)$.

Binomialheapar

- Baserade på positionellt talsystem.
- $5 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2$.
- En binomialheap av storlek 5 består av 2 heapordnade binomialträd:
  - Ett med storlek 2^0.
  - Ett med storlek 2^2.
- Ett heapordnat binomialträd av storlek 2^k består av:
  - En rot ($\leq$ värden i övriga noder).
  - En binomialheap av storlek $2^k - 1$.

Binomialheapar

- Tom kö: Inga träd.
- `find-min`: Minsta roten.
- `merge`: ”Addera” köerna.
- `insert`: Skapa kö av storlek ett, kör `merge`.
- `delete-min`: Ta bort minsta roten, kör `merge`.
- `decrease-key`: Ändra prioritet, bubbla.

Binomialheapar

Tidskomplexiteter (värsta fallet, $O(1)$-jämförelser):

- Tom kö: $O(1)$.
- `find-min`: $O(\log n)$ ($O(1)$).
- `merge`: $O(\log n)$.
- `insert`: $O(\log n)$.
- `delete-min`: $O(\log n)$.
- `decrease-key`: $O(\log n)$
  (om positionen är känd).

Binomialheapar

Amorterade tidskomplexiteter
($O(1)$-jämförelser, enkeltrådad användning):

- `merge`: $O(\log n)$.
- `insert`: $O(1)$.
- `delete-min`: $O(\log n)$.

Binomialheapar

Bevisskiss:

- Ett mynt på varje binomialträd (varje bit $= 1$).
- `merge`, `delete-min`: $O(\log n)$ faktisk tid, måste betala (max) $O(\log n)$ mynt.
- `insert`: $1 + 101001111 = 101010000$.
  Varje etta betalar för sig själv,
  återstår $O(1)$ arbete och betalning av 1 mynt.

Vad är tidskomplexiteten för följande program, där `i++` körs n gånger?
Använd det logaritmiska kostnadskriteriet.

```
int i = 0;
i++;
i++;
...
i++;
```

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

Tidskomplexiteter

	Binär heap	Braun-heap	Binomial-heap
`find-min`	$O(1)$	$O(1)$	$O(1)$
`delete-min`	$O(\log n)$	$O(\log n)$	$O(\log n)$
`insert`	$O(1)$ (medel)	$O(\log n)$	$O(1)$ (am)
`decrease-key`	$O(\log n)$	$O(n)$	$O(\log n)$
`merge`	$O(n)$	$O(n)$	$O(\log n)$

Andra prioritetsködatastrukturer

Jämförelse på Wikipedia.

Sammanfattning

Prioritetsköer:

- Binära heapar.
- Braunheapar.
- Binomialheapar.

Nästa gång:

- Hashtabeller.