# Föreläsning Datastrukturer (DAT037)

Nils Anders Danielsson

2015-11-13

# Idag

Prioritetsköer:

- Binära heapar.
- Heapsort.
- Leftistheapar.
- Maxifobiska heapar.

# Värstafalls-tids-komplexitet

# Värstafallstidskomplexitet

- En viss algoritm kan ta olika lång tid för olika indata som har samma storlek.
- Värstafallstidskomplexiteten anger hur snabb algoritmen är i det värsta fallet för (t ex) indata av en viss storlek:

$$T_{\textsf{worst case}}(n) = \sup \{ T(x) \mid \textsf{size}(x) = n \} .$$

(Kan vara odefinierad.)

# Värstafallstidskomplexitet

En fråga från förra föreläsningen var dåligt formulerad. Bättre:

Om man implementerar prioritetskö-ADTn med **länkade** listor, vad blir **värstafalls**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`: $\Theta(n)$.

Kunde också använt $O$ istället för $\Theta$.

# 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 $\Theta(\log n)$.

# 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.
- `modify-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).
- `modify-key`: $O(\log n)$.

(Givet att jämförelser tar konstant tid.)

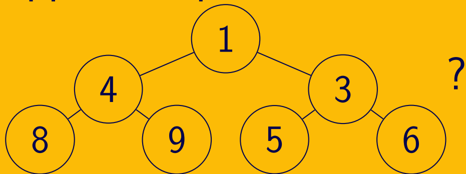
De flesta noderna är långt ned i trädet.
Medelkomplexitet för `insert`: $O(1)$.

# 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 $i$s vänstra barn: $2i$.
- Nod $i$s högra barn: $2i + 1$.
- Nod $i$s förälder ($i > 1$): $\lfloor i/2 \rfloor$.

Vad blir resultatet om `delete-min` appliceras på




?

A:

| 3 | 4 | 5 | 6 | 8 | 9 |
|---|---|---|---|---|---|

B:

| 3 | 5 | 6 | 4 | 8 | 9 |
|---|---|---|---|---|---|

C:

| 3 | 4 | 8 | 9 | 5 | 6 |
|---|---|---|---|---|---|

D:

| 3 | 4 | 5 | 8 | 9 | 6 |
|---|---|---|---|---|---|

# `modify-key`

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

Kan använda extra datastruktur.

Exempel: Hashtabell.

# `build-heap`

`build-heap`: Konverterar lista/array/... till heap.

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

Heapordningsegenskapen uppfylls
(kan bevisas med induktion).

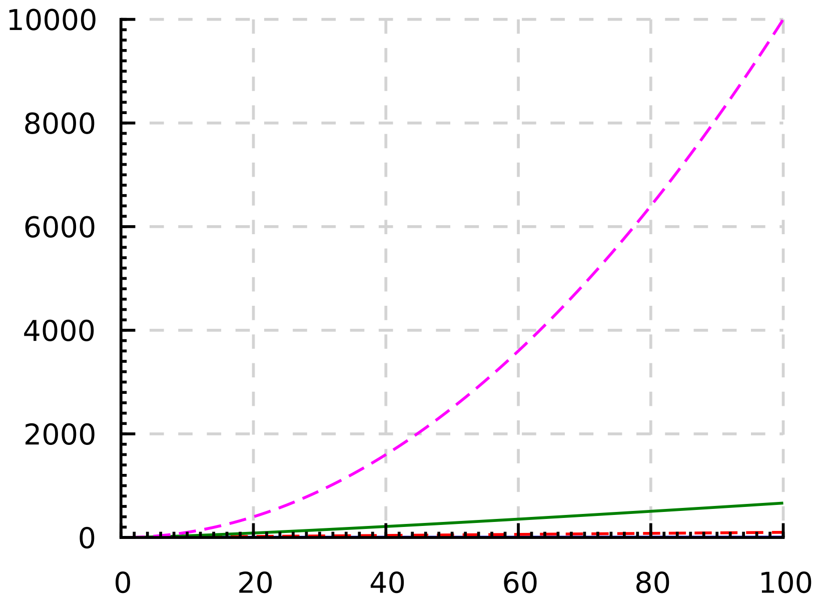
# `build-heap`: tidskomplexitet

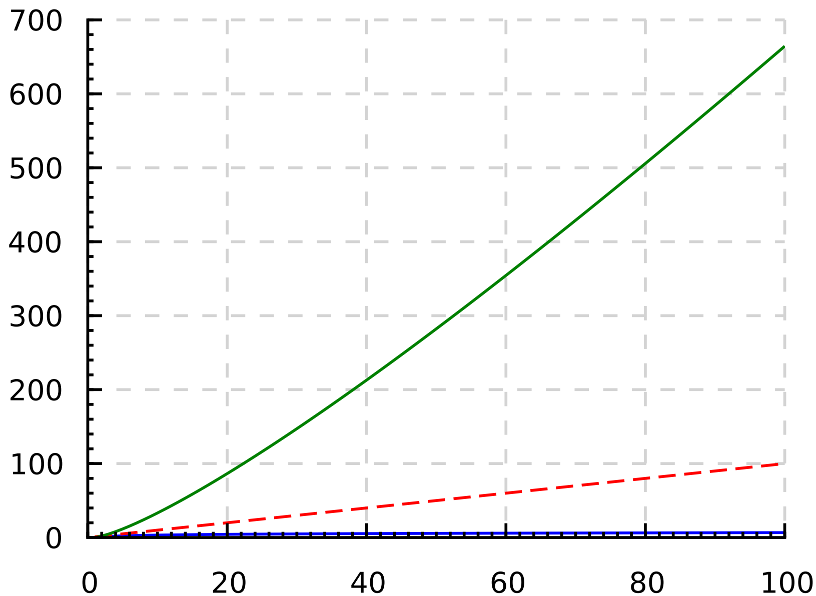
- Tid för viss nod: $O(\text{nodens höjd})$.
  (Givet $O(1)$-jämförelser.)
- Total tid: $O(\text{summan av alla höjder})$.
- Nästan alla noder är långt ned.
- Total tid: $\Theta(n)$.
- Bevis: Se boken.

# Heapsort

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

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


10000
8000
6000
4000
2000
0
0
20
40
60
80
100
$\log_2 n$
$n$
$n \log_2 n$
$n^2$



700
600
500
400
300
200
100
0
0
20
40
60
80
100
log₂ n
n
n log₂ n

# Leftistheapar

# `merge`

- Det verkar inte gå att slå ihop två binära heapar, implementerade med arrayer, på ett effektivt sätt.
- Med *leftistheapar*: $O(\log n)$ (med $O(1)$-jämförelser).
- Leftistheapar, implementerade i Haskell, är också persistenta.

# Leftistheapar

- Heapordnade (ofta pekarbaserade) binära träd, med extra invariant (senare).
- Grundläggande operation: `merge`.
- Lätt att implementera `insert`, `delete-min` med hjälp av `merge`.

# Leftistheapar

```
-- Invariant för Node x l r:
-- * x är mindre än eller lika med
--   alla element i l och r.
data PriorityQueue a
  = Empty
  | Node a (PriorityQueue a) (PriorityQueue a)

empty :: PriorityQueue a
empty = Empty

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

# Leftistheapar

```
insert :: Ord a =>
          a -> PriorityQueue a -> PriorityQueue a
insert x t = merge (Node x Empty Empty) t

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

-- Precondition: Kön får inte vara tom.
deleteMin :: Ord a =>
             PriorityQueue a -> PriorityQueue a
deleteMin Empty        = error "deleteMin: Tom kö."
deleteMin (Node _ l r) = merge l r
```

# Leftistheapar

```
merge :: Ord a =>
         PriorityQueue a -> PriorityQueue a ->
         PriorityQueue a
merge Empty              r                 = r
merge l                  Empty             = l
merge l@(Node xl ll rl) r@(Node xr lr rr) =
  if xl <= xr then
    Node xl ? ?  -- Ska använda ll, rl och r.
  else
    Node xr ? ?  -- Ska använda l, lr och rr.
```

```
Node xl ? ?  -- Ska använda ll, rl och r.
```

## Vilka två heapar ska slås ihop?

- Beskriv egenskapen hos de två heapar som ska slås ihop.
- Mål: $O(\log n)$ rekursiva anrop till `merge`.
- Tjuvtitta inte på nästa slide.

# Maxifobiska heapar (sidospår)

Om vi slår ihop de två minsta heaparna:

- Andelen noder i de två heaparna $\leq \frac{2}{3}$.
- Låt $M(n)$ vara största möjliga antalet anrop till `merge` då vi slår ihop två heapar med total storlek $n$:

$$M(0) = 1, M(1) = 1$$

$$M(n) \leq 1 + M\left(\left\lfloor \frac{2(n-1)}{3} \right\rfloor\right), \text{om } n > 1$$

$$\Rightarrow M(n) = O(\log n)$$

# Maxifobiska heapar (sidospår)

Tar tid att beräkna storleken.

- Kan spara storlek i noder.

# Leftistheapar

## Null path length

- -1 för tomma träd.
- Annars: Antal steg från roten
  till närmaste noden med max ett barn.

```
npl :: PriorityQueue a -> Integer
npl Empty          = -1
npl (Node _ l r) = 1 + min (npl l) (npl r)
```

# Leftistheapar

## Null path length

- -1 för tomma träd.
- Annars: Antal steg från roten
  till närmaste noden med max ett barn.

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

# Leftistheapar

- De första $1 + \texttt{npl}\ \ t$ nivåerna måste vara fulla:

$$\texttt{size}\ \ t \geq 2^{1+\texttt{npl}\ \ t} - 1.$$

  (Enkelt induktionsbevis.)
- Alltså:

$$\texttt{npl}\ \ t = O(\log n),$$

  där $n$ är trädets storlek.

# Leftistheapar

Om vi slår ihop heaparna med minst npl:

- Summan av de två heaparnas npl minskar med minst ett för varje rekursivt anrop.
- Till att börja med
  (om heaparna har total storlek $n$):

$$\texttt{npl l} + \texttt{npl r} = O(\log n).$$

- Minsta möjliga npl-summa: $-2$.
- Antalet anrop till `merge`: $O(\log n)$.

# Leftistheapar

Leftistheapar implementeras dock på ett annat sätt. `merge` traverserar högerryggraderna:

```
merge :: Ord a =>
         PriorityQueue a -> PriorityQueue a ->
         PriorityQueue a
merge Empty               r                 = r
merge l                   Empty             = l
merge l@(Node xl ll rl) r@(Node xr lr rr) =
  if xl <= xr then
    Node xl ll (merge rl r)
  else
    Node xr lr (merge l rr)
```

(Inte färdigoptimerad!)

# Leftistheapar

- Träd kan vara väldigt obalanserade:
  inga vänsterbarn, bara högerbarn.
- Det gör `merge` linjär (i värsta fallet).
- Lösning: Se till att högerryggraden är kort.

# Leftistheapar

## Leftist (invariant)

För `Node x l r`, `npl l` $\geq$ `npl r`.

Om invarianten gäller:

- Antal noder på högerryggraden:

$$1 + \texttt{npl t} = O(\log n),$$

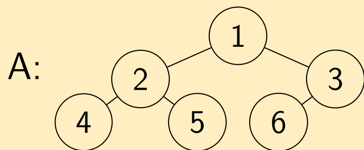
  där $n$ är `t`s storlek.
- Alltså är högerryggraden kort.

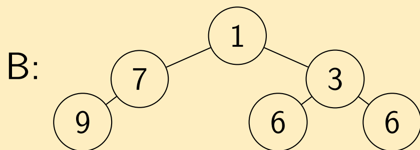
# Leftistheapar

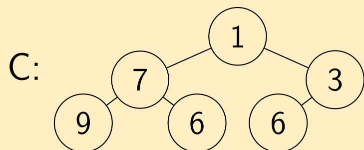
## Leftistheapinvarianter

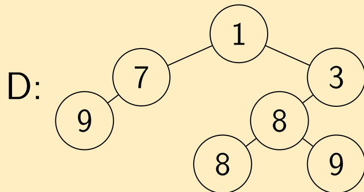
1. Heapordnad.
2. Leftist.

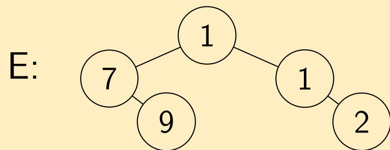
# Identifiera leftistheaparna.

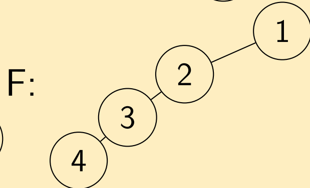
A:



B:



C:



D:



E:



F:


# Leftistheapar

- Den föregående implementationen av `merge` bryter ibland leftistinvarianten.
- Enkel lösning:
  Byt plats på det vänstra och det högra delträdet.

# Leftistheapar

Gammal kod:

```
merge :: Ord a =>
         PriorityQueue a -> PriorityQueue a ->
         PriorityQueue a
merge Empty               r                 = r
merge l                   Empty             = l
merge l@(Node xl ll rl) r@(Node xr lr rr) =
  if xl <= xr then
    Node xl ll (merge rl r)
  else
    Node xr lr (merge l rr)
```

# Leftistheapar

Ny kod:

```
merge :: Ord a =>
         PriorityQueue a -> PriorityQueue a ->
         PriorityQueue a
merge Empty              r                  = r
merge l                  Empty              = l
merge l@(Node xl ll rl) r@(Node xr lr rr) =
  if xl <= xr then
    node xl ll (merge rl r)
  else
    node xr lr (merge l rr)
```

# Leftistheapar

Smart konstruerare som ser till att leftistinvarianten gäller:

```
-- Precondition för node x l r:
-- * x är mindre än eller lika med
--   alla element i l och r.
node :: a -> PriorityQueue a -> PriorityQueue a ->
        PriorityQueue a
node x l r =
  if npl l >= npl r then
    Node x l r
  else
    Node x r l
```

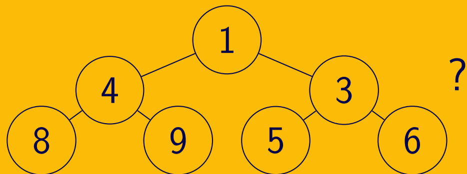
# Leftistheapar

En sista förändring:

- Den rekursiva beräkningen av `npl` är onödig.
- Istället: Spara `npl`-värden i noder.

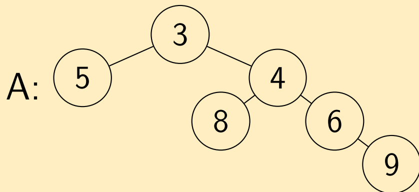
```
-- Invarianter: ...
data PriorityQueue a
  = Empty
  | Node Integer a (PriorityQueue a)
                   (PriorityQueue a)

npl :: PriorityQueue a -> Integer
npl Empty          = -1
npl (Node n _ _ _) = n
```

Vad är resultatet av att applicera `deleteMin` på

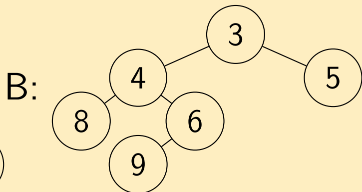



?

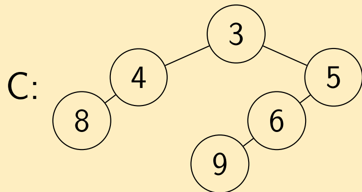
A:



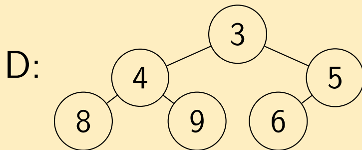

B:




C:




D:



# `modify-key`

Om datastrukturen inte kan uppdateras:

- Leta efter nod: $O(n)$.
- Bubbla upp eller ned: $O(n)$.

Om vi har en lämplig uppdateringsbar datastruktur och vi känner till nodens position:

- `modify-key` kan implementeras med tidskomplexiteten $O(\log n)$.

I båda fallen: Om jämförelser tar konstant tid.

# build-heap

- En binär heap är en leftistheap.
- Med lämplig uppdateringsbar datastruktur: Kan använda ungefär samma algoritm som för binära heapar.

# build-heap

$\Theta(n)$-implementation som fungerar även om datastrukturen inte kan uppdateras:

```
fromList :: Ord a => [a] -> PriorityQueue a
fromList xs =
  mergeAll [ node x empty empty | x <- xs ]
  where
  mergeAll []  = empty
  mergeAll [q] = q
  mergeAll qs  = mergeAll (mergePairs qs)

  mergePairs (l : r : qs) = merge l r :
                            mergePairs qs
  mergePairs [q]          = [q]
  mergePairs []           = []
```

# Prioritetsköer, samman-fattning

# Tidskomplexiteter

| | Binär heap | Leftistheap (icke modifierbar) |
|---|---|---|
| `find-min` | $\Theta(1)$ | $\Theta(1)$ |
| `delete-min` | $O(\log n)$ | $O(\log n)$ |
| `insert` | $\Theta(1)$ (medel) | $O(\log n)$ |
| `modify-key` | $O(\log n)$ | $O(n)$ |
| `merge` | $\Theta(n)$ | $O(\log n)$ |
| `build-heap` | $\Theta(n)$ | $\Theta(n)$ |

(Om jämförelser tar konstant tid.)

# Andra prioritetsködatastrukturer

Jämförelse på Wikipedia.

# Sammanfattning

Idag:

- Värstafallstidskomplexitet.
- Prioritetsköer.
  - Binära heapar.
  - Leftistheapar.
  - Maxifobiska heapar.

Nästa gång:

- Hashtabeller.
- Mer om sortering.