Föreläsning 4
Datastrukturer (DAT037)

Fredrik Lindblad[1]

8 november 2017

[1] Slides skapade av Nils Anders Danielsson har använts som utgångspunkt. Se http://www.cse.chalmers.se/edu/year/2015/course/DAT037

Innehåll

- Mängd och avbildning
- Prioritetskö
- Binär heap
- Leftist-heap

Mängd (Set) och avbildning (Map)

ADT	Operationer
Set	add(x), remove(x), contains(x)
Map	put(k, e), remove(k), containsKey(k), get(k)

En mängd är ett specialfall av en avbildning, men avbildningar är i princip lika svåra att implementera än mängder. De implementeras på samma sätt. I Java har vi `TreeSet` och `TreeMap`, `HashSet` och `HashMap`.

Mängder

ADT:

- Konstruerare för tom mängd.
- `add`(x): Lägger till x till mängden.
  Mängder innehåller ej dubbletter.
- `contains`(x): Avgör om x finns i mängden.
- `remove`(x): Tar bort x från mängden.
- Gränssnitt i Java: `Set`

Avbildningar/maps

ADT:

- Konstruerare för tom avbildning.
- `put(`k`, `v`)`: Lägger till bindningen $k \mapsto v$, mellan en nyckel och ett värde.
  Om det finns en gammal bindning $k \mapsto v'$ skrivs den (t ex) över.
- `get(`k`)`: Om det finns en bindning $k \mapsto v$ så ges v som svar.
- `remove(`k`)`: Tar bort bindningen $k \mapsto v$ (om det finns en sådan bindning).
- Gränssnitt i Java: `Map`

Krav på nyckeltypen

Olika mängd-/avbildningsdatastrukturer har olika krav på nyckeltypen:

- Likhetstest (`equals`)
- Olikhetstest (`Comparable`, `Comparator`)
- Hashfunktion (`hashCode`)

Alla kräver likhetstest.

Exempel: Avbildning

```
Map<String, Integer> m = new HashMap<>();
m.put("Agnes", 576);
m.put("Arvid", 340);
m.put("Agnes", 616);
m.remove("Arvid");
System.out.println(m.containsKey("Arvid"));
  // false
System.out.println(m.containsKey("Agnes"));
  // true
System.out.println(m.get("Agnes"));
  // 616
```

Prioritetsköer

Prioritetsköer

Köer där varje element har viss prioritet.

Gränssnitt (exempel):

- Konstruerare för tom kö.
- `insert(x)`: Lägger till element.
- `find-min()`: Ger tillbaka minsta elementet.
- `delete-min()`:
  Tar bort och ger tillbaka minsta elementet.
- `remove(x)`: Tar bort ett element.
- `merge(q)`: Slår ihop två köer.

Prioritetsköer

Några tillämpningar:

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

Labb 2: Implementera och tillämpa prioritetskö. Man brukar låta mindre värde innebära högre prioritet, d.v.s. `delete-min()` ger det (eller ett av de) minsta elementet i kön.

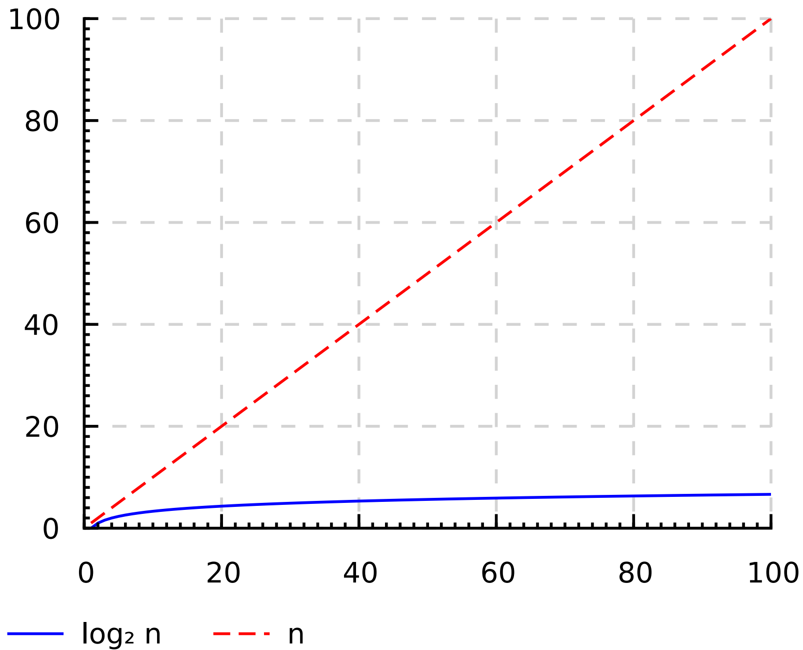
Om man implementerar prioritetskö-ADTn med listor, vad blir tidskomplexiteten (ev amorterad) för `insert` och `delete-min`?

- A – `insert`: $\Theta(1)$, `delete-min`: $\Theta(1)$
- B – `insert`: $\Theta(1)$, `delete-min`: $\Theta(n)$
- C – `insert`: $\Theta(n)$, `delete-min`: $\Theta(1)$
- D – `insert`: $\Theta(n)$, `delete-min`: $\Theta(n)$

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

- A – `insert`: $\Theta(1)$, `delete-min`: $\Theta(1)$
- B – `insert`: $\Theta(1)$, `delete-min`: $\Theta(n)$
- C – `insert`: $\Theta(n)$, `delete-min`: $\Theta(1)$
- D – `insert`: $\Theta(n)$, `delete-min`: $\Theta(n)$

Svar: B och C. B om man lagrar elementen osorterat och C om man lagrar dem sorterat i stigande prioritet.


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.

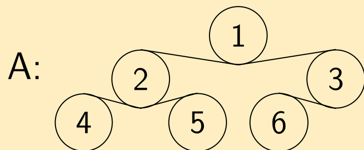
Komplett binärt träd

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

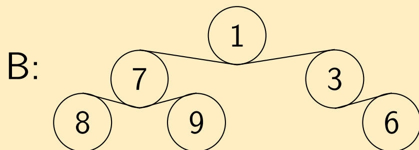
En binär heap med n noder har höjden $\Theta(\log n)$.

Identifiera alla binära heapar.

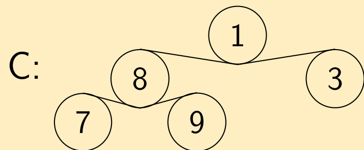
A:



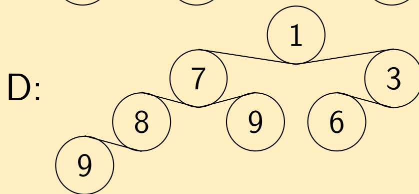
B:



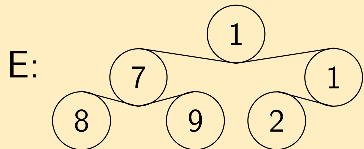
C:



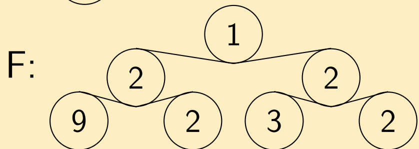
D:



E:

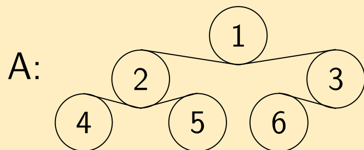


F:

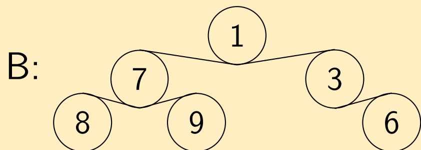


Identifiera alla binära heapar.

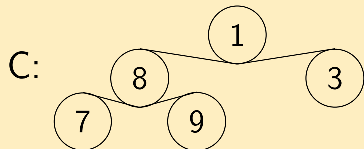
A:



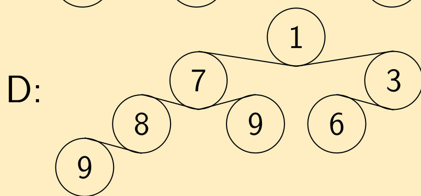
B:



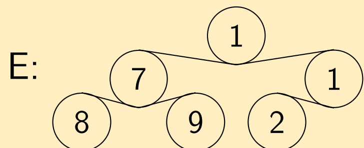
C:



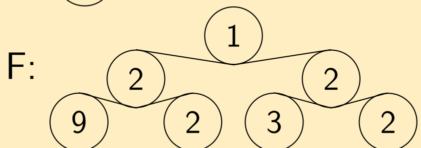
D:



E:



F:



Svar: A, E, F

Binär heap implementerar prio. kö

- 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.
- `remove`: Let upp elementet, ersätt med sista elementet.
  bubbla åt rätt håll.

Bubbla upp/ned tills heapordningsegenskapen återställts.

Bubbla upp/ned

- Om värdet är mindre (prio är högre) än i noden ovan, byt plats och kolla rekursivt upp tills ordningen är rätt.
- Om värdet är större (prio är lägre) än i någon av noderna nedan, byt plats med barnet som är minst och fortsätt rekursivt i denna gren.

Tidskomplexitet

Om man kan hitta sista noden och föräldrar snabbt:

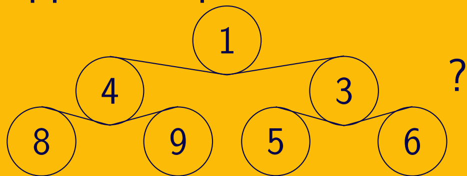
- Tom kö: $\Theta(1)$.
- `find-min`: $\Theta(1)$.
- `insert`: $O(\log n)$ (kanske amorterat).
- `delete-min`: $O(\log n)$ (kanske amorterat).
- `remove`: $O(n)$.

Implementation av binära heapar

Man brukar representera trädet med array (labb 2).
Det passar mycket bra för kompletta träd.

- Roten på position 0.
- Sista elementet på position $n - 1$.
- Första tomma cellen på position n.
- Nod is vänstra barn: $2i + 1$.
- Nod is högra barn: $2i + 2$.
- Nod is förälder ($i > 0$): $\lfloor (i - 1)/2 \rfloor$.

Vad blir resultatet om `delete-min` appliceras på



A:

3	4	5	6	8	9

B:

3	5	6	4	8	9

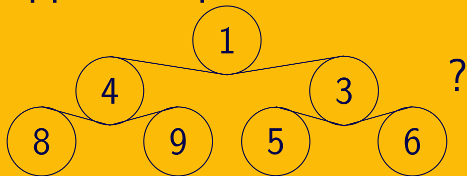
C:

3	4	8	9	5	6

D:

3	4	5	8	9	6

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

Svar: D

`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 som har löv.
  (Kan hoppa över alla löv.)

Heapordningsegenskapen uppfylls
(kan bevisas med induktion).

`build-heap`: tidskomplexitet

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

Leftistheapar

merge

- Det verkar inte gå att slå ihop två binära heapar, implementerade med arrayer, på ett effektivt sätt.
- Med *leftistheapar*: $O(\log n)$

Leftistheapar

- Heapordnade (ofta pekarbaserade) binära träd, som inte är kompletta utan uppfyller annan balanseringsinvariant, leftist träd-egenskapen.
- Grundläggande operation: `merge`.
- Lätt att implementera `insert`, `delete-min` med hjälp av `merge`.

Null path length

För en nod, X, i ett träd är $\mathrm{npl}(X)$ (null path length of X) kortaste vägen till en nod med max ett barn. (Detta är motsatsen till nodens höjd.) För tomt träd är null path length -1.

Null path length

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

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

  (Enkelt induktionsbevis.)
- Alltså:

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

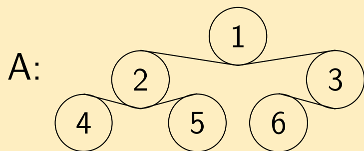
  där n är trädets storlek.

Leftist-träd-egenskapen

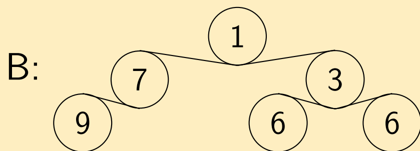
För varje nod i trädet gäller för dess vänster- och högerbarn, l och r, att $\mathrm{npl}(l) \geq \mathrm{npl}(r)$.
Detta medför att i leftist-träd har den högraste vägen max $\lfloor \log(n+1) \rfloor$ noder, där n är totalt antal noder.

Identifiera leftistheaparna.

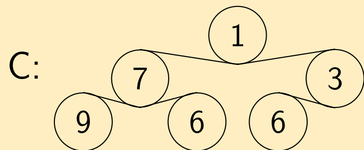
A:



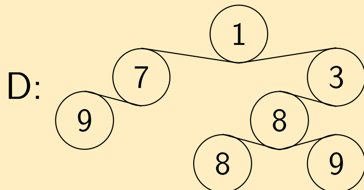
B:



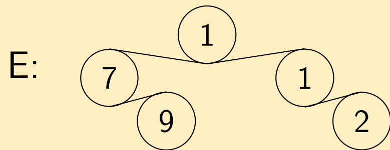
C:



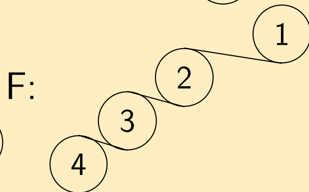
D:



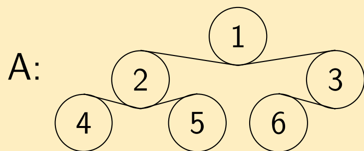
E:

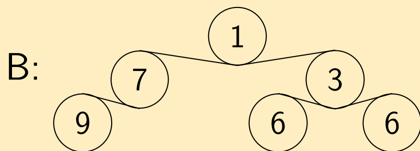


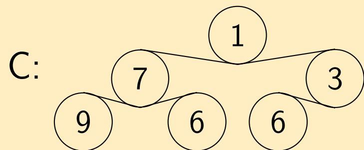
F:

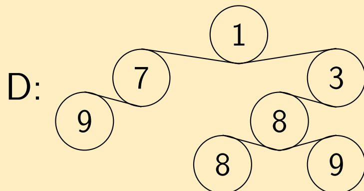


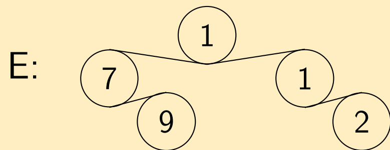
Identifiera leftistheaparna.

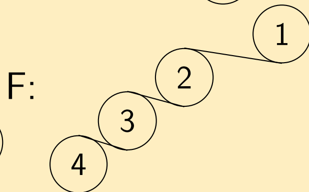
A:


B:


C:


D:


E:


F:


Svar: A, D, F

Merge

Behöver lagra npl för varje nod.

Rekursiv implementering:

- Om något av träden är tomma, ta det andra trädet.
- Låt $t_{\leq}$ vara trädet med minst rot och $t_{>}$ det andra. Slå ihop $t_{\leq}.h$ (höger delträd) med $t_{>}$ (rekursivt anrop).
- Ersätt $t_{\leq}.h$ med det trädet som är resultatet av det rekursiva anropet. Detta får vi anta (induktion) är en leftist heap med alla element som förekommer i $t_{\leq}.h$ och $t_{>}$. Kalla det nya trädet t.

Merge

- Eftersom vi valde delträd av det med minst rot vet vi att det nya delträdet (rotens högerdelträd) uppfyller heap-ordningen gentemot föräldern. Hela t uppfyller alltså heap-ordningen.
- Men kanske inte leftist-egenskapen. Om $\mathrm{npl}(t.v) < \mathrm{npl}(t.h)$ så byt helt enkelt plats på barnen. Låt t' beteckna det resulterande trädet (som kan vara identiskt med t).
- Slutligen uppdatera npl i roten. Den är $\mathrm{npl}(t'.h) + 1$.

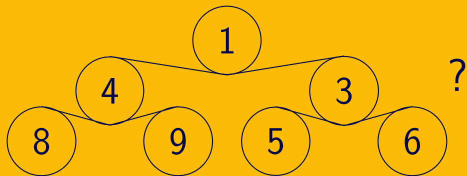
Tidskomplexitet för merge

Om vi anropar merge för t_1 och t_2 så är antalet rekursiva anrop begränsat av antalet noder i högraste vägen i t_1 + antalet noder i högraste vägen i t_2. Dessa är $O(\log(|t_1|))$ resp. $O(\log(|t_2|))$. Varje ekvering av den rekursiva operationen (exklusive det rekursiva anropet) tar konstant tid. Varje rekursivt anrop tar oss ett steg ner på den högraste vägen i antingen t_1 eller t_2. Alltså är komplexiteten för merge $O(\log(|t_1|) + \log(|t_2|)) = O(\log(\max(|t_1|, |t_2|))$, d.v.s. $O(\log(n))$ om n är storleken på den största heapen.

Implementering av insert och deleteMin

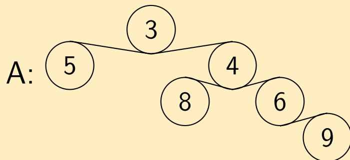
- Insert implementeras genom att slå ihop trädet med ett träd bestående av ett element (det nya). Tidskomplexitet: $O(\log(n))$.
- deleteMin genom att ersätta roten med hopslagningen av vänster och höger delträd. Tidskomplexitet: $O(\log(n))$.

Vad är resultatet av att applicera `deleteMin` på

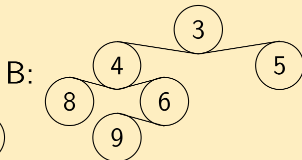


?

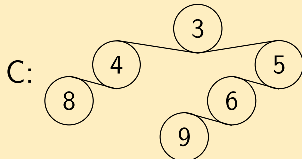
A:



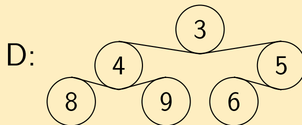
B:



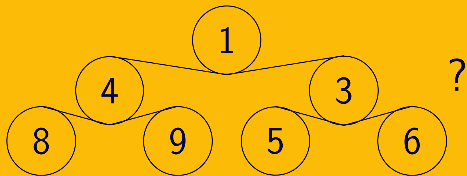
C:



D:

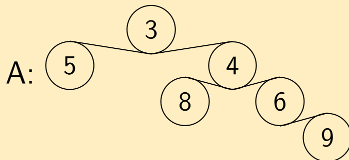


Vad är resultatet av att applicera `deleteMin` på

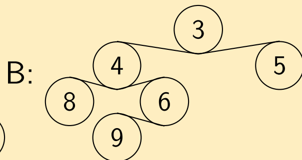


?

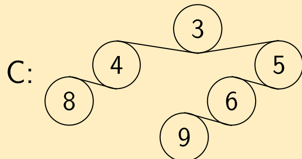
A:



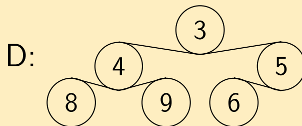
B:



C:



D:



Svar: B

Prioritetsköer, samman- fattning

Tidskomplexiteter

	Binär heap (array)	Leftistheap
`find-min`	$O(1)$	$O(1)$
`delete-min`	$O(\log n)$ am	$O(\log n)$
`insert`	$O(\log n)$ am	$O(\log n)$
`merge`	$\Theta(n)$	$O(\log n)$
`build-heap`	$\Theta(n)$	$\Theta(n)$