22
Hobe.

Binære hobe.

Minimum-hob og maximum-hob.

Den abstrakte datatype minimum-hob.

Opbygning af hobe.

Operationen siv-ned.

Indsættelse i hobe.

Sletning af minimalt element i hobe.

Repræsentation.

PS1 -- Hobe

Noter

Binære hobe.

Definitioner:

- En **binær minimum-hob** er et binært træ, som opfylder følgende:
  - Struktur-egenskab: Træet er et komplet binært træ.
  - Hob-egenskab: For *enhver* knude N, og de to knuder N1 og N2, som er rod i venstre og højre undertræ af N, gælder at

    N <= N1 og N <= N2.

    forudsat N1 og N2 eksisterer.

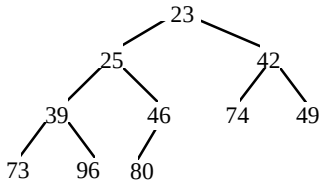
- En **binær maximum-hob** er defineret tilsvarende, blot med N >= N1 og N >= N2.
- Et **komplet binært træ** er et binært træ, som
  - er helt fyldt op (indeholder flest mulige knuder)
  - pånær evt. rækken af blade, som fyldes op fra venstre mod højre.

Noter

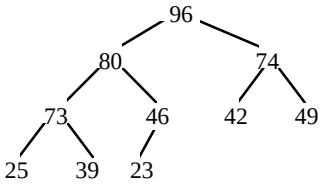
På næste slide giver vi eksempler på såvel en minimum-hob som en maximum-hob. De to træer på næste slide er selvsagt også eksempler på komplette, binære træer.

Eksempler på binære hobe.

Minimum-hob:



Maximum-hob:



PS1 -- Hobe

Noter

Den abstrakte datatype 'minimum-hob'.

Operationers syntaks	Uformel semantik	Køretid
Lav-minimum-hob(L: Liste): Hob	Lav en hob ud fra elementerne i listen L, og returner hoben.	O(n) n er antallet af elementer i listen L.
Indsæt(e: element, H: hob)	Indsæt e i hoben H, således at H forbliver en hob.	O(log n) n er antallet af elementer i hoben H.
Slet-minimum(H: hob): element	Slet det mindste element i H, og sørg for, at de resterende elementer udgør en hob. Returner det slettede element.	O(log n)
Hob-knude(n: integer): Knude	Returner knuden med et bestemt nummer i H. Knuder nummereres fortløbende fra roden (nummer 1) og vandret fra venstre mod højre gennem træets niveauer.	O(1)

PS1 -- Hobe

Noter

De tre øverste operationer ovenfor er de basale operationer på en minimum-hob. Vi kunne naturligvis let formulere ovenstående som en egentlig klasse HOB med operationerne

create(L: Liste)

indsæt(e: element)

slet_minimum

minimum_element: element.

Vi har dog valgt at følge den mere konventionelle stil fra kapitlerne om søgning.

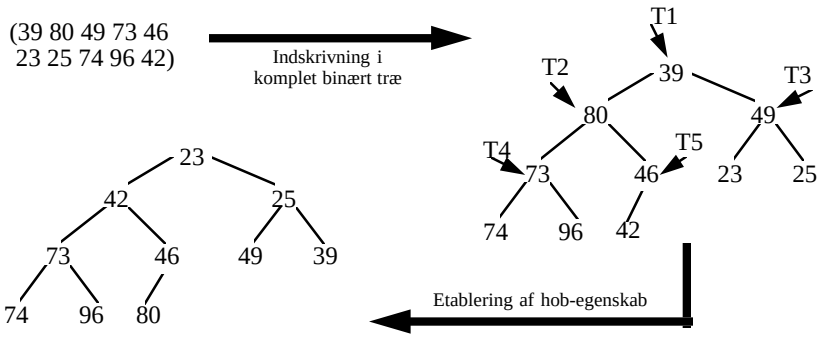
Operationen Hob-knude er en praktisk operation, som giver en nummerering af knuderne i en hob. I forbindelse med array-repræsentationen af en hob, som vi introducerer tilsidst i dette kapitel, bliver det gjort helt præcist, hvad Hob-knude(n) er for en knude.

Opbygning af hobe.

```
Lav-minimum-hob(L: Liste): Hob
   Indskriv L's elementer i et komplet, binært træ T ;
   for  alle undertræer U  af T
     from Tn to T1
   do  Siv-ned(U) ;
   Returner T
```

Køretid:

O(n), n = længde(L)



PS1 -- Hobe

Noter

På næste slide vises en algoritmeskitse for proceduren Siv-ned, som er helt central under opbygningen af hoben.

Nederst på denne slide er vist et eksempel på opbygningsprocessen. En liste indskrives i et komplet binært træ. Effekten af for-løkken er vist med pilen "Etablering af hob-egenskab".

Som det fremgår, gennemløber vi alle undertræer af T på en ganske bestemt måde for at danne en hob. Det er selvfølgelig kun nødvendigt at gennemløbe de træer, som har et eller flere ikke-tomme undertræer. (Et træ, som ikke har undertræer, er altid en hob).

I træet til højre har vi vist listen T5 .. T1 af træer, der kommer i betragtning, når det skitserede binære træ skal laves til en hob.

Bemærk, at vi i dette gennemløb af træer (i eksemplet fra T5 mod T1) får opfyldt Siv-ned's prebetingelse, der jo siger, at undertræer af parameteren i Siv-ned skal være hobe.

Opbygning af en maximum-hob, **lav-maximum-hob**, foregår ved nøjagtig samme metode. Blot skal siv-ned operationen tage hensyn til, at vi nu ønsker at bygge en maximum-hob.

Om tidskompleksiteten af hob opbygning: Tidskompleksiteten af lav-minimum-hob (og lav-maximum-hob, som er tilsvarende) er bemærkelsesværdig. Arbejdet ved opbygningen ses let at være propertionalt med summen af alle knuders højde. (For en definition af knuders højde, se forelæsningen om træer). Denne sum kan vises at være O(n). Der henvises til Weiss for detaljer.

Operationen 'siv-ned'.

Siv-ned(T: Binært-træ):
Præbetingelse: De to evt. undertræer af T er begge minimum-hobe.

```
If T har mindst ét undertræ
then let  N be roden i T, og K nøglen i N.
          T1 be venstre undertræ af T, med rod N1 og nøgle K1.
          T2 be højre undertræ af T, med rod N2 og nøgle K2.
     in
       if not ( K <= K1 and K <= K2)
       then  if K1 < K2
             then  ombyt  N1 og N  ;  -- T1's rod er nu N
                   Siv-ned(T1)
             else  ombyt  N2 og N  ;  -- T2's rod er nu N
                   Siv-ned(T2) ;
```

Postbetingelse: T er en minimum-hob

Køretid af 'Siv-ned':

O(log n), n = antallet af knuder i T.

Noter

Ovenstående udgave af siv-ned er beregnet på minimum-hobe. For at håndtere maximum-hobe skal følgende tests "**if not** (K <= K1 **and** K <= K2) **then** **if** K1 < K2" erstattes af "**if not** (K >= K1 **and** K >= K2) **then** **if** K1 > K2 ...".

I ovenstående algoritme er der et specialtilfælde, som vi ikke har taget i betragtning: N kan være rod i et træ, som ikke har et højre undertræ. Derved findes T2 og N2 ikke. I dette tilfælde skal vi blot sammenligne K og K1, og ombytte N1 og N hvis ikke K <= K1.

Indsættelse i hobe.

Indsæt(e: element, T: hob):

Præbetingelse: T er en minimum-hob

let N **be** *en ny knude med indhold e.*

Indsæt N på næste plads i T, så det udvidede træ stadig er komplet **;**
Siv-op(T, N) ;

Postbetingelse: T er en minimum-hob

Siv-op(T: binært-træ, N: knude):

If not (N er rod i T)

let K_N **be** nøglen af N
R **be** far af N
K_R **be** nøglen af R

if $K_N < K_R$
then *ombyt N og R* ; --N er nu far af R
Siv-op(T, N)

Køretid af 'Indsæt' og 'Siv-op':

O(log n) hvor n = antallet af knuder i T

PS1 -- Hobe

Noter

Vi har valgt at give en rekursiv skitse af proceduren Siv-op. Dette giver en god og enkel forståelse af algoritmen, som iøvrigt ikke behøver at være særlig meget langsommere og ret meget mere lagerkrævende end en ren iterativ løsning. Årsagen er, at Siv-op er halerekursiv.

Sletning af det minimale element fra minimum-hobe.

```
Slet-minimum(H: hob): element
  pre-betingelse: H er en minimum-hob
  Let R be roden af H
      n be antal af elementer i H
      N be Hob-knude(n)

    Returner indholdet af R ;
    Slet knude N fra H ;
    {H er stadigvæk en minimum-hob}
    Erstat knude R med N ;
    {H er nu et komplet binært træ med rod N}
    Siv-ned(H);
  post-betingelse: H er en minimum-hob
```

Køretid:

O(log n)

hvor n = antal elementer i H

Noter

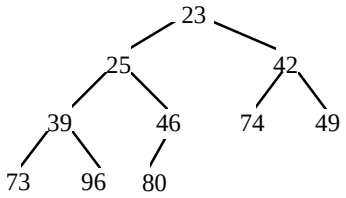
Ovenstående algoritmeskitse af Slet-minimum er formuleret som en funktion med sideeffekt. Dette er egentlig udtryk for dårlig programmeringsstil, men det er en hensigtsmæssig og kompakt udtryksmåde for, at vi ønsker at udtrække det minimale element fra hoben, og samtidig reetablere hob-egenskaben blandt de resterende elementer.

Læg mærke til pre- og postbetingelserne af Slet-minimum. Hvis denne operation -- sammen med de øvrige hob operationer -- udgjorde en egentlig dataabstraktion (en klasse), ville pre- og postbetingelserne helt naturligt have udgjort klasse-invarianten.

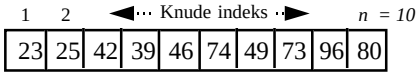
Repræsentation af hobe.

Det er mere hensigtsmæssigt at repræsentere en hob i et array end i en kædet træstruktur.

Array-repræsentationen er attraktiv, idet en hob altid er et komplet binært træ.



En minimum-hob.



Repræsentation af hoben.

Træ-operationer:

Antag at knude K har indeks *i* :

Venstre-søn(K) har indeks *2i.*
Højre-søn(K) har indeks *2i + 1.*
Far(K) har indeks *i div 2.*

Noter

Hvis en knude N har indeks *n* i array repræsentationen af hoben H, da gælder at Hob-knude(*n*) = N.