Ugeseddel 10

Litteratur:

- CLRS kapitel 12 dog ikke 12.4, dog ikke bevis for theorem 12.1
- CLRS kapitel 17, dog er 17.3 kursorisk og analysedele baseret på potentialmetoden (17.3) i kapitel 17.4 skippes. Dvs. i 17.4.1 skippes den sidste del af teksten startende med "We can use the potential…" og i 17.4.2 skippes den sidste del af teksten startende med "We can now use the potential…..".

Noter:

- 10.1 binere soegetraeer noter CLRS kap 12
- 10.2-3 Amortiseret analyse
- 10.2-3 Dynamiske tabeller noter CLRS 10 og 17

Mål for ugen:

- Træer
- Binære søgetræer
- Amortiseret analyse og metoder (dog er potentialmetoden kursorisk).

Plan for ugen:

- Tirsdag bliver brugt på træer og specielt søgetræer
- Torsdag bliver forelæsningerne primært brugt på amortiseret analyse. Men der kan forsat arbejdes delvist med søgetræer til øvelserne.

Opgaver i træer og binære søgetræer:

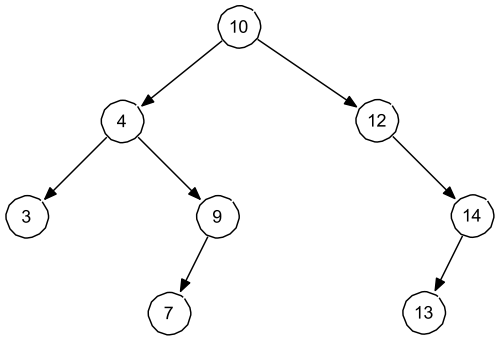
Opgaverne er i denne uge opdelt i emner frem for dage, opgaverne i træer og binære søgetræer er ment til øvelsestimerne tirsdag, mens amortiseret analyse er til torsdag. Der er rigeligt opgaver i træer så man kan bruge eventuelt resterende tid torsdag til at løse dem man mangler. Desuden er der et helt afsnit med ekstra opgaver i træer.

Opgaver markeret med (ekstra) kan gemmes til resten af opgaverne er regnet. I bunden af ugesedlen finder du nogle svære opgaver markeret med stjerne hvis du mangler udfordring eller er færdig med dagens opgaver.

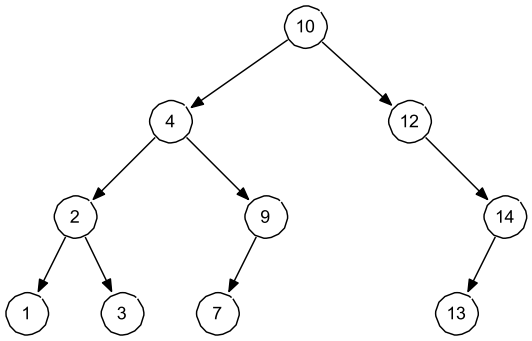
Pas på: Opgaver markeret med "*" er svære, "" er meget svære, og "***" har du ikke en chance for at løse.**

1. **Opvarmning**

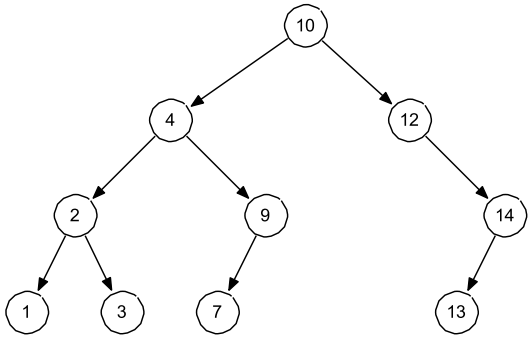
1.1 Angiv hvordan det binære søgetræ nedenfor ser ud efter indsættelse af elementet med nøgle 8



1.2 Angiv hvordan det binære søgetræ nedenfor ser ud efter sletning af elementet med nøgle 4.

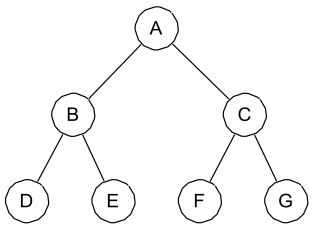


1.3 Angiv rækkefølge af knuderne som de bliver skrevet ud ved preorder gennemløb af træet



2 Trægennemløb

2.1 Betragt nedenstående træ.

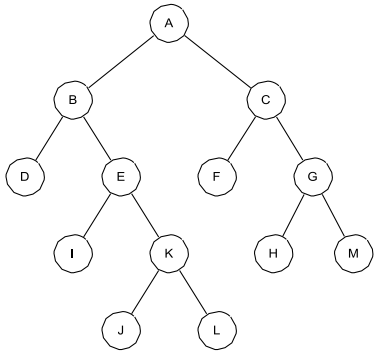


Hvilke af følgende sekvenser af bogstaver bliver udskrevet ved et preorder, inorder og postorder gennemløb af ovenstående træ.

1 A B C D E F G	2 G F E D C B A	3 A B D E C F G
4 D B E A F C G	5 D E B F G C A	6 C B D A F E G

De næste to opgaver omhandler rodfæstede binære træer. Hver knude har *enten to eller ingen* børn. Knuden x's venstre barn betegnes left[x], og dens højre barn betegnes right[x]. Hvis knuden x ikke har nogle børn, har left[x] og right[x] den specielle nil værdi. Hvis knude x ikke har nogle børn, kaldes den et blad. Ellers kaldes den en indre knude. Såfremt rodknuden for et træ er nil, er træet tomt.

2.2 Betragt nedenstående træ.



Hvad er længden af den korteste hhv. længste rod-til-blad sti i træet

2.3 Giv en rekursiv algoritme KortesteSti(x) der givet rodknuden til et binært træ returner længden af den korteste rod-til-blad sti i træet. Angiv gerne algoritmen i pseudokode. Angiv køretiden af din algoritme i asymptotisk notation og argumenter for at den er korrekt.

3. Blade og højde Lad T være et binært træ med n knuder og rod v.
3.1 Giv en rekursiv algoritme, der givet v beregner antallet af blade i T. Skriv pseudokode for din løsning.
3.2 Giv en rekursiv algoritme, der givet v beregner højden af T. Skriv pseudokode for din løsning.

4 Gennemløb af binære søgetræer
4.1 Giv en algoritme, der givet et binært træ T med en nøgle i hver knude, afgør om T overholder søgetræsinvarianten.

5 Perfekt balancerede binære søgetræer
5.1 Tegn et binært søgetræ med nøglerne (1, 2, …, 7) således at træet har højde 2. Angiv en rækkefølge af tallene således at Tree-Insert algoritmen fra CLRS vil konstruere dit træ, hvis udført på tallene i den rækkefølge.
5.2 Lad A være en sorteret tabel af $n = 2^{h+1} - 1$ forskellige tal (et perfekt balanceret træ med højde h har præcist $2^{h+1} - 1$ knuder). Giv en sekvens af indsættelser af tallene i A i et binært søgetræ T således at T bliver et komplet binært træ af højde h.

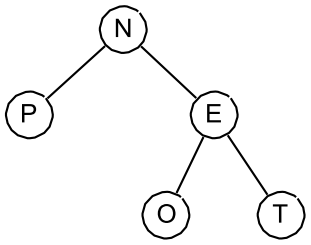
6 Mere rekursion på træer Denne opgave handler om rekursion og rodfæstede binære træer.
Hver knude har *enten to eller ingen* børn. Knuden x's venstre barn betegnes *left*[x], og dens højre barn betegnes *right*[x]. Hvis knuden x ikke har nogle børn, har *left*[x] og *right*[x] den specielle nil værdi. Hvis knude x ikke har nogle børn, kaldes den et blad. Ellers kaldes den en intern knude. Såfremt rodknuden for et træ er nil, er træet tomt. Til hver knude i træet er der knyttet en farve; knuden x har farven *farve*[x].

Betragt følgende algoritme:

Algorithm 3 Udskriv(x)

```
if (x ≠ nil)
  print farve[x]

  Udskriv(left[x])
  Udskriv(right[x])
```



Figur 1: Træ med knuder, der har farver angivet som bogstaver.

Lad x være rodknuden for træet i figur 1. Et kald til algoritmen Udskriv(x) vil da udskrive farvesekvensen "NPEOT".

6.1 Du skal ændre algoritmen **Udskriv**, således at det rekursive gennemløb ved kaldet til **Udskriv**(x) for rodknuden x i træet fra figur 1 udskriver farvesekvensen "NETOP" i stedet. Argumenter for at din algoritme er korrekt.

Med den definition af binære træer, vi benytter i denne opgave, kan vi beregne antallet af knuder og antallet af blade i et træ ved hjælp af følgende to algoritmer:

Algorithm 4 Nodes(x)

```
if (x = nil) return 0
else return 1 + Nodes(left[x]) +Nodes(right[x])
```

Algorithm 5 Leaves(x)

```
if (x = nil) return 0
else if left[x] = nil return 1
else return Leaves(left[x]) + Leaves(right[x])
```

6.2 En indre knude i et træ er en knude, som ikke er et blad. Konstruer en algoritme Intern(x), der givet en rodknude x for et træ returnerer antallet af interne knuder i træet. Angiv tidskompleksiteten/køretiden af din løsning i O-notation.

6.3 Et træ har en blå rodvej, hvis der er en vej i træet fra roden x til et blad, hvor alle knuderne på vejen fra x til bladet har farven blå. Konstruer en rekursiv algoritme **BlåRodvej**(x), der returnerer tallet 1, hvis træet med roden x har en blå rodvej. Hvis en sådan vej ikke

eksisterer, skal algoritmen returnere 0. Argumenter for at din algoritmen er korrekt. Angiv tidskompleksiteten/køretiden af din løsning i O-notation. Begrund dit svar.

7 CLRS opgaver

7.1 Hvor findes element med hhv. mindste og største nøgle i et binært søgetræ?
7.2 CLRS 12.1-1.
7.3 CLRS 12.2-5. *Hint:* argumenter vha. modstrid.
7.4 CLRS 12.2-6. Hint: argumenter vha. modstrid.
7.5 CLRS 12.2-7
7.6 CLRS 12.3-3

8 Rekursion på træer. Denne opgave handler om rekursion og rodfæstede binære træer. Hver knude har *enten to eller ingen* børn. Knuden x's venstre barn betegnes $left[x]$, og dens højre barn betegnes $right[x]$. Hvis knuden x ikke har nogle børn, har $left[x]$ og $right[x]$ den specielle nil værdi. Hvis knude x ikke har nogle børn, kaldes den et blad. Ellers kaldes den en indre knude. Såfremt rodknuden for et træ er nil, er træet tomt. Hver knude i træet har et felt $size[x]$, der indeholder et heltal.

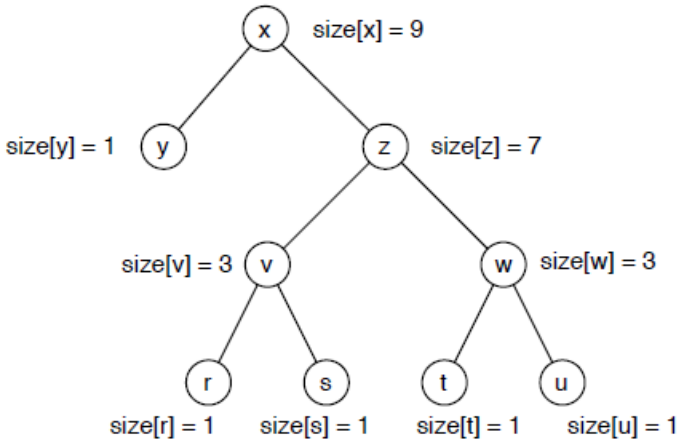
Betragt følgende algoritme:

Algorithm 3 Zero(x)

```
1: if (x ≠ nil) then
2:   size[x] = 0
3:   Zero(left[x])
4:   Zero(right[x])
```

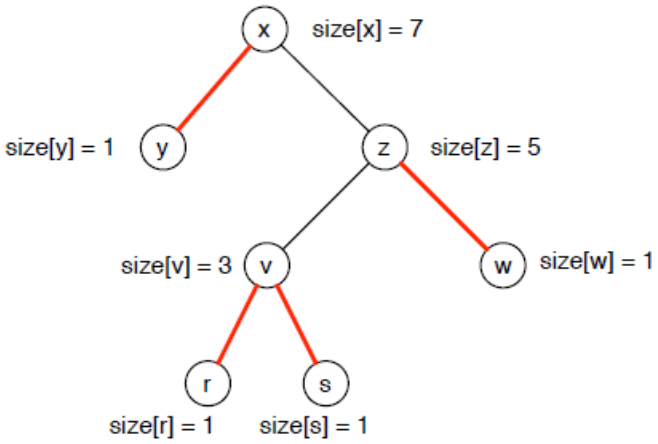
Lad x være rodknuden for træet T. Efter udførelsen af proceduren Zero(x) vil alle felterne $size[x]$ være 0.

8.1 Angiv køretiden af proceduren Zero(x) i O-notation, hvor x er roden i et træ med n knuder. Begrund dit svar.
8.2 I de følgende opgaver betegner $T(x)$ undertræet med rod x i T. Proceduren InitSize(x), skal givet roden x i et træ T med n knuder initialisere felterne $size[y]$ for alle knuder y i T, så size[y] bliver lig med antallet af knuder i $T(y)$. Se eksemplet nedenfor.



Giv pseudokode for InitSize(x), så køretiden er $O(n)$.

8.3 For et træ T siger vi at en kant (u, v), hvor u er forælder til v, er *rød*, hvis antallet af knuder i undertræet $T(u)$ er mindst dobbelt så stort som antallet af knuder i $T(v)$. Dvs. efter udførelse af InitSize gælder $size[u] \geq 2size[v]$ for alle *røde* kanter (u,v) i T. Se eksempel nedenfor.



Konstruer en rekursiv algoritme RødKant(x) der givet en rodknude x for et træ returnerer antallet af *røde* kanter i træet. Angiv tidskompleksiteten/køretiden af din løsning i O-notation.

8.4 [*] Angiv i O-notation en øvre grænse for antallet af røde kanter, der kan være på stien fra en knude til roden i et træ T.

Opgaver i amortiseret analyse:

1. Amortiseret analyse

1.1 CLRS 17.1-1
1.2 CLRS 17.1-2
1.3 CLRS 17.1-3
1.4 CLRS 17.2-2
1.5 CLRS 17.2-1
1.6 CLRS 17.2-3

2. Lad A være et array af heltal. Vi siger, at et tal A[i] er højre-maksimalt, hvis det er større end alle tal A[j] for j > i. Følgende algoritme returnerer alle højre-maksimale tal i A. Hvad er køretiden for algoritmen? *Hint: læg mærke til at stakken består af indekser og ikke værdier.*

Algorithm 6 FindHøjreMax(A)

```
Lad S være en stak
for i = 0 to n-1:
    while (S not empty and A[i] > A[top(S)]):
        Pop(S)
    Push(S, i)
Return S
```

3. Forestil dig, at vi har en datastruktur, hvor den i'te operation tager f(i) tid (vi vil definere f(i)

herunder). Lad os udføre n operationer, som altså koster f(1) + f(2) + … + f(n). For hver af følgende definitioner af f(i) afgør hvad den amortiserede køretid af en operation er:

3.1 f(i) = i^2 hvis i er en toerpotens og f(i) = 1 ellers.

3.2 f(i) = i hvis i er ulige og f(i) = 1 ellers.

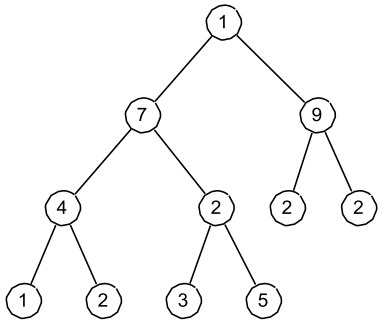
4. Kø med stakke CLRS 10.1-6.

Ekstra opgaver

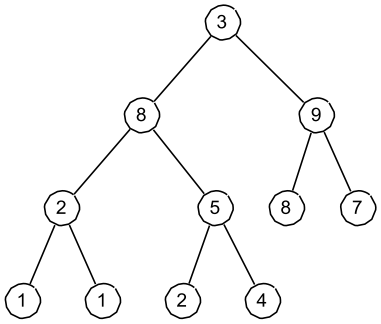
1. Vægtbalanceret træ Denne opgave omhandler rodfæstede binære træer. Hver knude har *enten to eller ingen* børn. Knuden *x*'s venstre barn betegnes *left*[*x*], og dens højre barn betegnes *right*[*x*]. Hvis knuden *x* ikke har nogle børn, har *left*[*x*] og *right*[*x*] den specielle `nil` værdi. Hvis knude *x* ikke har nogle børn, kaldes den et blad. Ellers kaldes den en indre knude. Såfremt rodknuden for et træ er `nil`, er træet tomt. Til hver knude i træet er der knyttet en vægt; knuden *x* har vægten *weight*[*x*]. Bemærk at vægten ikke skal forstås som en nøgle og træerne ikke er søgetræer.

1.1 **Ens vægte** Giv en effektiv algoritme EnsVægt(x) der givet rodknuden, x, til et vægtet binært træ returnerer 1 hvis der er mindst to knuder i træet der har samme vægt og 0 ellers. Angiv køretiden af din algoritme og argumenter for at den er korrekt.

1.2 To knuder i et træ er søskende, hvis de har samme forælder. Dvs. *left*[*x*] og *right*[*x*] er søskende. Vi siger at en knude *v* i et vægtet binært træ er *vægtbalanceret* hvis vægten af *v* højst er dobbelt så meget som vægten af dens søsken. Et træ er vægtbalanceret hvis alle knuder i træet er vægtbalancerede. Hvilke af nedenstående træer er vægtbalancerede?



(a)

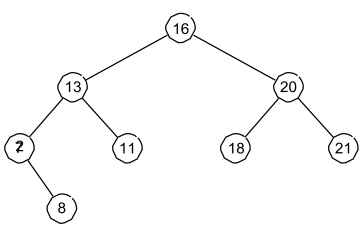


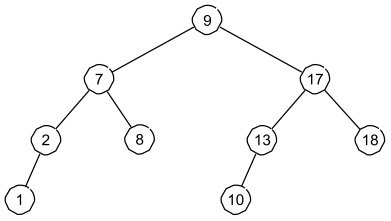
(b)

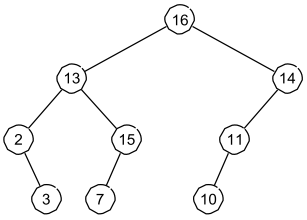
1.3 Skriv pseudokode for en *rekursiv* algoritme VægtBalance(*x*), der givet rodknuden, x, til et vægtet binært træ der returnerer 1 hvis træet er vægtbalanceret og 0 ellers.
Angiv køretiden af din algoritme og argumenter for at den er korrekt.

2. Binære søgetræer

2.1 Hvilken af følgende træer er et binært søgetræ?







(a) (b) (c)

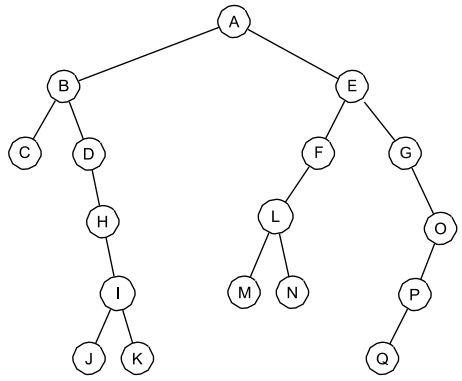
2.2 Angiv preorder-, inorder- og postordersekvensen af nøgler for træet i (b)

3. CLRS

3.1 CLRS 12.1-5
3.2 CLRS 12.2-1
3.3 CLRS 12.2-2
3.4 CLRS 12.2-3
3.5 CLRS 12.3-1
3.6 CLRS 12.3-2
3.7 CLRS 12.3-6

4. Mere rekursion i træer. Denne opgave omhandler rodfæstede binære træer. Enhver knude x har felterne key[x], left[x] og right[x], der betegner hhv. nøglen, det venstre barn og det højre barn for x.

4.1 En knude i et binært træ er en kædeknude hvis knuden har netop et barn og det barn ikke er et blad. Angiv alle kædeknuder i nedenstående træ.



4.2 Giv en algoritme KÆDEKNUDE(X), der givet en knude x returnerer sand hvis og kun hvis x er en kædeknude. Din algoritme skal køre i konstant tid. Skriv din algoritme i pseudokode. Du må gerne antage at du har en konstant tids algoritme LEAF(X), der givet en knude x, returnerer sand hvis og kun hvis x er et blad.

4.3 Giv en rekursiv algoritme TÆLKÆDEKNUDER(X), der givet rodknuden returnerer antallet af kædeknuder i træet. Skriv din algoritme i pseudokode og analyser køretiden af din algoritme som funktion af n, hvor n er antallet af knuder i træet.

Stjerneopgaver (svære ekstraopgaver):

1. [*] **Dynamiske træer:** Lad der være givet en funktion Fjern(F,e) som fjerner en kant e fra et træ i en skov F. Lad e forbinde knuderne x og y, og efter e er fjernet bliver knuden x roden i sit eget træ: Tx og knuden y bliver roden i sit eget træ Ty. Antag at Fjern har køretid linær i antal af knuder i det mindste af træerne Tx og Ty

1.1. Lad F initielt have n knuder (og dermed højst n-1 kanter). Vis at de højst n-1 Fjern operationer samlet tager O(n log n) tid.

1.2. Giv en implementering af Fjern(F,e) med den angivet køretid og funktion.

2. **[**]** Lad A være et array af heltal. Vi siger, at A er 2-tæt, hvis det gælder, at A[i] og A[i+1] er inden for en faktor 2 af hinanden for alle i. F.eks. er array'et [1,2,3,2,4,7,4] 2-tæt, men [1,2,4,2,5] er ikke (fordi 5 og 2 ikke er inden for en faktor 2 af hinanden). Betragt følgende algoritme:

Algorithm 7 2-tæt(*A*)

```
for i = 0 to n-2:
    if A[i+1] * 2 < A[i]:
        A[i+1] = ceil(A[i] / 2)
for i = n-1 to 1:
    if A[i-1] * 2 < A[i]:
        A[i-1] = ceil(A[i-1] / 2)
Return A
```

2.1. Argumenter (løst) for at algoritme 7 sørger for at A er et 2-tæt array.

2.2. Argumenter for, at hvis summen af A er m inden et kald til algoritme 7, så er summen O(m) efter algoritmen har kørt.

3. **[**] Dynamiske tabeller og stakke**

Vi er interesseret i at implementere en stak ved hjælp af dynamiske tabeller uden at fastsætte en maksimal størrelse på tabellen til at starte med. Det kan antages at det at allokere og deallokere en tabel tager konstant tid. Løs følgende opgaver:

3.1. Generaliser dynamiske tabeller til også at kunne håndtere stakke, der "skrumper" undervejs (dvs. understøtter både PUSH og POP operationer). Køretiden af enhver operation skal være $\Theta(n)$ og til ethvert tidspunkt skal din løsning bruge lineær plads i antallet af elementer i stakken.

3.2. [**] Samme som 3.1 men køretiden af enhver sekvens af n operationer skal være $\Theta(n)$.

3.3. [**] Vis hvordan man kan opnå $O(1)$ amortiseret tid per stak operation med dynamiske tabeller og lineær plads i antallet af elementer i stakken. Kig kun på stakke der vokser og ignorer "skrump". *Hint:* Tænk på hvordan man kan fordele arbejdet.

3.4 [**] Samme som 3.3 men nu skal stakken også kunne skrumpe.

4. **[**] Kombination af en stak og en kø**

Du skal implementere en datastruktur, som er en kombination af en stak og en kø. Den skal understøtte operationerne:

- enqueue(S, x): Tilføj x til S.
- dequeue(S): Fjern det element, der har været længst tid i S.
- pop(S): Fjern det element, som har vret kortest tid i S.

Vi kan nemt implementere denne datastruktur vha. en hægtet liste L, der har en peger, L.tail, til bagerste element i listen. Da implementeres enqueue og pop ved hhv. at tilføje og fjerne et element forrest i listen, og dequeue implementeres ved at fjerne det bagerste element i listen. Med denne implementation er køretiden for alle tre operationer O(1).

Udfordringen er at implementere denne datastruktur ved hjælp af 3 stakke, således at hver operation tager O(1) amortiseret tid.

Hint: I har tidligere implementeret en almindelig kø ved at bruge to stakke, således at begge kø-operationer var O(1) amortiseret. Overvej hvorfor din læsning til den opgave IKKE længere er O(1) amortiseret, når vi også må fjerne det element som har været kortest tid i køen. Hvordan kan en ekstra stak afhjælpe det problem, som opstår?

Bemærkninger:

Flere af opgaverne er inspireret af opgaver stillet af Philip Bille og Inge Li Gørtz i kurset Algoritmer og Datastrukturer, på DTU, http://www2.compute.dtu.dk/courses/02105+02326/2015/#generelinfo