

Notater: INF2220

Veronika Heimsbakk  
veronahe@student.matnat.uio.no

13. oktober 2014

## **Innhold**

# Introduksjon

Dette er notater til kurset INF2220–Algoritmer og datastrukturer ved Universitetet i Oslo. Disse notatene er i all hovedsak basert på forelesningsfoiler, egne notater og læreboka. Brukes til repetisjon før eksamen og som notater under eksamen. Merk at notatene helt sikkert inneholder feil og mangler.

## 1 Kjøretid

Ettersom kjøretiden til algoritmer som regel er et veldig komplekst uttrykk, så holder det at denne estimeres. Denne typen estimering kalles **asymptotic analysis**, eller **big-O** notasjon.

DEFINISJON 1.1. La  $f$  og  $g$  være funksjoner  $f, g : \mathcal{N} \rightarrow \mathcal{R}^+$ . Sier at  $f(n) = O(g(n))$  hvis det eksisterer positive heltall  $c$  og  $n_0$  slik at for hvert heltall  $n \geq n_0$ ,

$$f(n) \leq cg(n).$$

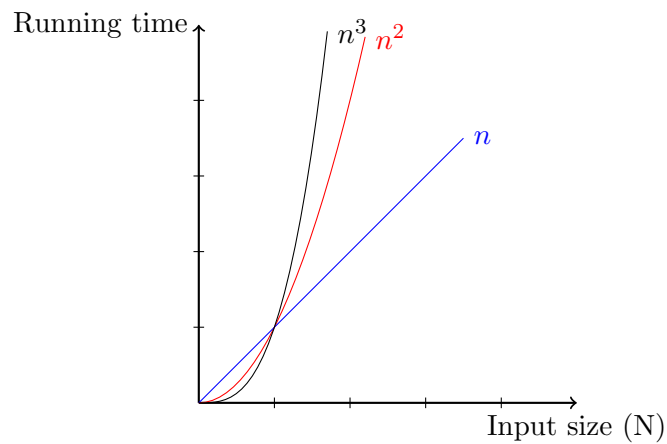
Når  $f(n) = O(g(n))$ , sier vi at  $g(n)$  er **upper bound** for  $f(n)$ , eller mer presist at  $g(n)$  er **asymptotic upper bound** for  $f(n)$ .

Funksjon	Navn
1	Konstant
$\log n$	Logaritmisk
$n$	Lineær
$n \log n$	
$n^2$	Kvadratisk
$n^3$	Kubisk
$2^n$	Eksponensiell
$n!$	

Tabell 1: Vanlige funksjoner for  $O(n)$

Et eksempel på enkel beregning av kjøretid for kode er:

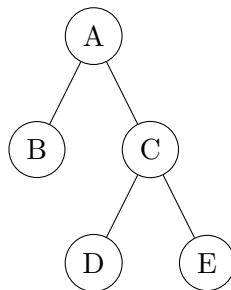
```
1 // O(n)
2 for (i = 0; i < n; i++)
3     array[i] = 0;
4
5 // O(n^2)
6 for (i = 0; i < n; i++)
7     for (j = 0; j < n; j++)
8         array[i][j] = 0;
```



Figur 1: Noen funksjoner for  $O(n)$ .

## 2 Trær

DEFINISJON 2.1. Et **tre** er en samling **noder**. Et ikke-tomt tre består av en **rot-node**, og null eller flere ikke-tomme **subtrær**. Fra roten går en **rettet kant** til roten i hvert subtre.



Figur 2: Eksempel på et tre.

I figur ??, så er A rot-noden. B og C er barna til A, og C er forelder til løvnodene D og E.

### 2.1 Traversering

**Pre-order** rot, venstre barn, høyre barn

**In-order** venstre barn, rot, høyre barn

**Post-order** venstre barn, høyre barn, rot

```

1  /* IN-ORDER TRAVERSAL */
2
3  public void inOrder (Node node) {
4      if (node != null) {
5          inOrder(node.left);
6          // do something with the node
7          inOrder(node.right);
8      }
9  }
10
11 /* PRE-ORDER TRAVERSAL */
12
13 public void preOrder (Node node) {
14     if (node != null) {
15         // do something with the node
16         preOrder(node.left);
17         preOrder(node.right);
18     }
19 }
20
21 /* POST-ORDER TRAVERSAL */
22
23 public void postOrder (Node node) {
24     if (node != null) {
25         postOrder(node.left);
26         postOrder(node.right);
27         // do something with the node
28     }
29 }

```

## 2.2 Terminologi

**Vei** Eller en *sti* fra node  $n_1$  til  $n_k$  er definert som en sekvens  $n_1, n_2, \dots, n_k$ , slik at  $n_i$  er forelder til  $n_{i+1}$  for  $1 \leq i \leq k$ .

**Lengden** Lengden av veien er antall *kanter* på veien;  $k - 1$ .

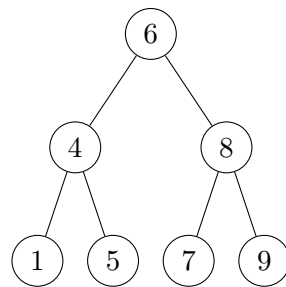
**Dybde** Definert av den unike veien fra roten til noden. Rotens dybde er 0.

**Høyden** Definert som lengden av den *lengste* veien fra noden til en løvnode. Alle løvnoder har høyde 0. Høyden til et tre er lik høyden til roten.

## 2.3 Binærtrær

I et binærtre er et tre der hver node aldri har mer enn to barn. I INF2220 er fokuset på *binære søketrær*. For binære søketrær skal følgende holde for hver node i treet:

- Alle verdiene i *venstre* subtre er *mindre* enn verdien i noden selv.
- Alle verdiene i *høyre* subtre er *større* enn verdien i noden selv.



Figur 3: Eksempel på et binært søketre.

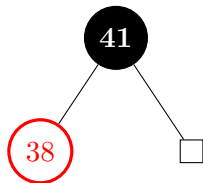
## 2.4 Rød-svarte trær

Et rød-svart tre er et binært søketre der hver node er farget enten rød eller svart slik at følgende holder:

1. Roten er svart.
2. Hvis en node er rød, så må barna være svarte.
3. Enhver sti fra en node til en null-peker må inneholde samme antall svarte noder.

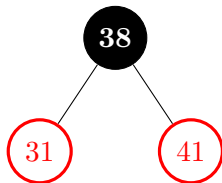
Dette sikrer at høyden på et slikt tre er maks  $2 \times \log_2(N + 1)$ .

1



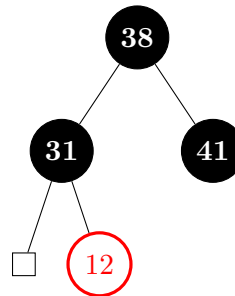
Setter inn 41 og 38 etter reglene.

2



Når 31 nå kommer inn, så må vi snu på treet og farge om 38 og 41.

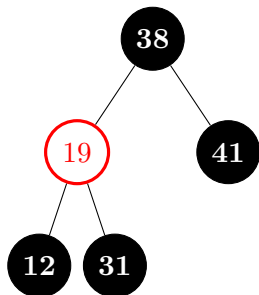
3



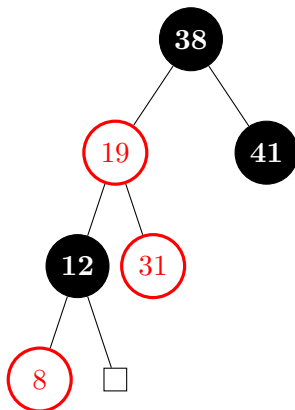
Siden løvnodene var røde når 12 skal inn, så må 31 og 41 farges om. 41 skal farges iht. regel nr. 3.

5.

4.



Når 19 nå kommer inn, så må vi igjen snu på treet for å opprettholde reglene til et BST. Farger nodene på nytt iht. reglene for rød-svarte trær.



Til slutt setter vi inn 8 som løvnode til 12. Her trenger vi ikke å snu eller fargelegge.

Figur 4: Eksempel på innsetting av tallene 41, 38, 31, 12, 19 og 8 i et rød-svart tre.

## 2.5 B-trær

## 3 Maps

## 4 Hashing

Idéen i hashing er å lagre alle elementene i en array (hashtabell) og la verdien i elementet  $x$  bestemme indeksen til  $x$  i hashtabellen. Egenskaper til en god hash-funksjon er at den er rask å beregne, kan gi alle mulige verdier fra 0 til tabellens størrelse  $-1$ . Og den gir en god fordeling utover tabellindeksene. En hashtabell tilbyr innsetting, sletting og søking med konstant tid.

### 4.1 Hash-funksjoner

**Eksempel** med heltall som nøkkel, begrenset antall tabellindekser. La hashfunksjonen være `hash(x, tableSize) = x mod tableSize`. Husk at hvis `tableSize` er 10 og alle nøklene slutter på 0, vil alle elementene havne på samme indeks. Huskeregel for dette: la alltid tabellstørrelsen være et primtall. Funksjonen under summerer verdiene til hver bokstav. Dette er en dårlig fordeling dersom tabellstørrelsen er stor.

```
1 int hash (String key, int tableSize) {
2     int hashValue = 0;
3     for (i = 0; i < key.length(); i++)
4         hashValue += key.charAt(i);
5     return (hashValue % tableSize);
6 }
```

### 4.2 Forskjellige typer hash-tabeller

- **Seperate chaining** er en hash-tabell hvor hver indeks peker til en liste av elementer.
- **Probing** er rommet mellom hver indeks.
  - *Lineær probing*; intervallene er satt (normalt 1).
  - *Kvadratisk probing*; intervallene øker med å legge til et kvadratisk polynom ved startverdi gitt av hashberegningen.

– *Dobbel hashing*; intervallene er gitt av en annen hash-funksjon.

Index	Linear probing	Quadratic probing	Separate chaining
0	9679	9679	
1	4371	4371	
2	1989		
3	1323	1323	1323 → 6173
4	6173	6173	4344
5	4344	4344	
6			
7			
8		1989	
9	4199	4199	4199 → 9679 → 1989

Tabell 2: Eksempel på hashing.

I tabell ?? har vi forskjellige hash-tabeller på tallene {4371, 1323, 4199, 4344, 9679, 1989} som skal settes inn i en tabell på størrelse 10, og med hash-funksjon  $H(X) = X \bmod 10$ .

**Lineær probing** her tar man tallet og setter inn i hash-funksjonen. Så for første tall (4371) vil indeksen bli 1. Hvis indeksen er opptatt, så plusser man på 1.

**Kvadratisk probing** er så og si det samme, men i stedet for å legge til 1 hvis indeksen er opptatt, så legger man på  $1^2, 2^2, 3^2, \dots$

**Dobbel probing** her lager vi en ny hash-funksjon når indeksen krasjer. Da tar man et tall  $P$ , som er det største primtallet som er mindre enn tabellstørrelsen. Dette brukes til å lage en ny hash-funksjon  $H(X) = R - (X \bmod R)$ .

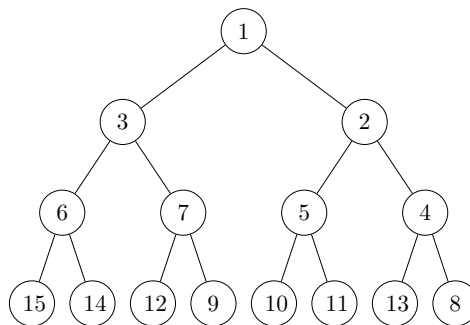
**Separate chaining** her legges det nye tallet på i en liste om indeksen er opptatt fra før.



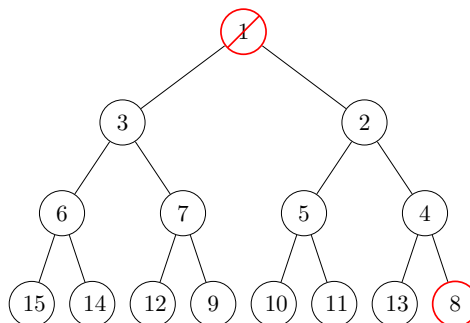
## 5 Prioritetskø

## 6 Heap

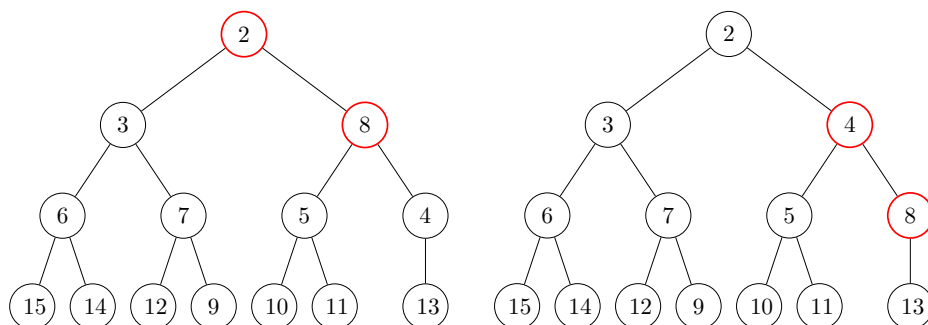
Skal gjøre operasjonen `deleteMin()` på følgende heap  $\times 3$ .



Begynner med å slette rot noden 1. Og flytter siste element (8) opp til rot posisjon.



Får da dette resultatet. Og vi må nå flytte 8 ned til riktig posisjon.



## 7 Grafer

### 7.1 Topologisk sortering

### 7.2 Dijkstra

### 7.3 Prim

### 7.4 Kruskal

### 7.5 Dybde-først

## 8 Kombinatorisk søk

## 9 Rekursjon

## 10 Tekstalgoritmer

**Tabeller**

**Figurer**