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

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

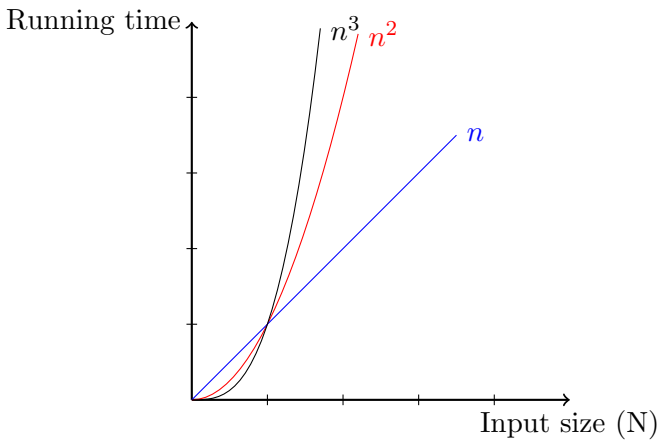
Når $f(n) = O(g(n))$, sier vi at $g(n)$ er ***upper bound*** *for $f(n)$, eller mer presist at $g(n)$ er* ***asymptotic upper bound*** *for $f(n)$.*

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:

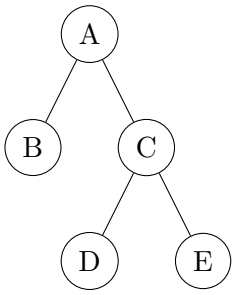
```
1 // O(n)
2 for (i = 0; i < n; i++)
3   array[i] = 0;
4
5 // O(n^2)
6 for (i = 0; i < n; i++)
7   for (j = 0; j < n; j++)
8     array[i][j] = 0;
```



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 2, 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

```java
/* IN-ORDER TRAVERSAL */

public void inOrder (Node node) {
  if (node != null) {
    inOrder(node.left);
    // do something with the node
    inOrder(node.right);
  }
}

/* PRE-ORDER TRAVERSAL */

public void preOrder (Node node) {
  if (node != null) {
    // do something with the node
    preOrder(node.left);
    preOrder(node.right);
  }
}

/* POST-ORDER TRAVERSAL */

public void postOrder (Node node) {
  if (node != null) {
    postOrder(node.left);
    postOrder(node.right);
    // do something with the node
  }
}
```

2.2 Terminologi

Vei Eller en *sti* fra node n_1 til n_k er definert som en sekvens $n_1, n_2, \ldots, n_k$, 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; $k - 1$.

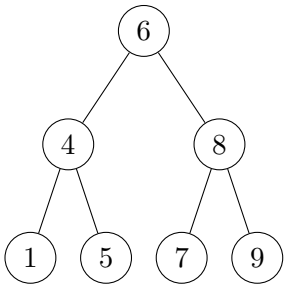
Dybde Definert av den unike veien fra roten til noden. Rotens dybde er 0.

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.

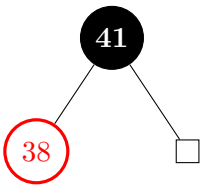
2.4 Rød-svarte trær

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.

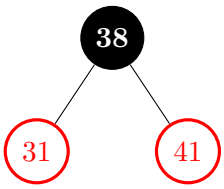
Dette sikrer at høyden på et slikt tre er maks $2 \times \log_2(N + 1)$.

1



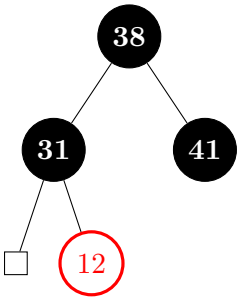
Setter inn 41 og 38 etter reglene.

2



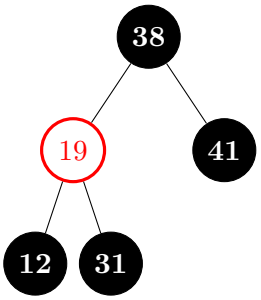
Når 31 nå kommer inn, så må vi snu på treet og farge om 38 og 41.

3



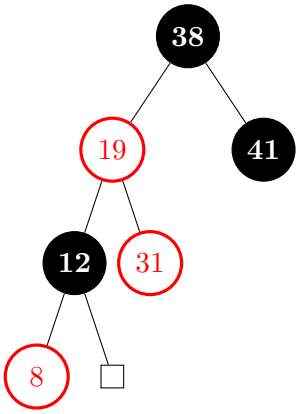
Siden løvnodene var røde når 12 skal inn, så må 31 og 41 farges om. 41 skal farges iht. regel nr. 3.

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.

5.



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 git 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, taleSize) = 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

- **Seprate 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	$1323 \rightarrow 6173$
4	6173	6173	4344
5	4344	4344	
6			
7			
8		1989	
9	4199	4199	$4199 \rightarrow 9679 \rightarrow 1989$

Tabell 2: Eksempel på hashing.

I tabell 4.2 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 mod 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 $1^2, 2^2, 3^2, \ldots$

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 mod R)`.

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

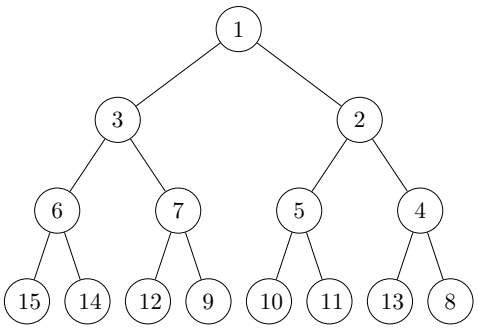
5 Heap

En **binær heap** er et komplett binærtre, hvor barna alltid er større eller lik sine foreldre. Og et komplett binærtre har følgende egenskaper:

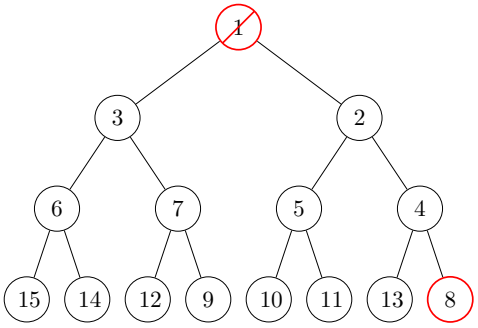
- Treet vil være i perfekt balanse.
- Løvnoder vil ha høydeforskjell på maksimalt 1.
- Treet med høyden h har mellom 2^h og $2^{h+1} - 1$ noder.
- Den maksimale høyden på treet vil være $\log_2(n)$.

5.1 Eksempel på sletting i min-heap

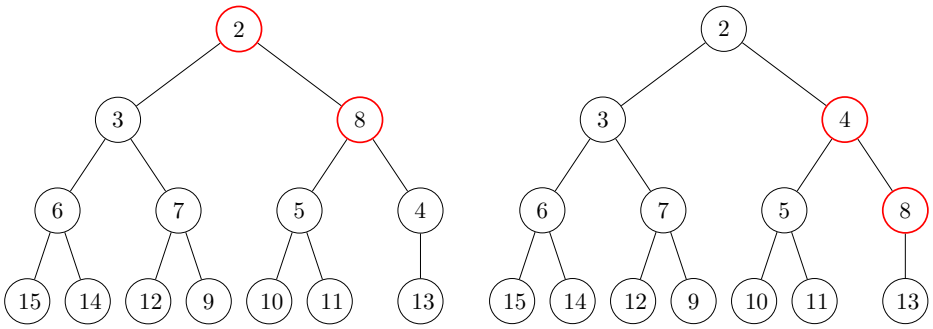
Skal gjøre operasjonen `deleteMin()` på følgende heap $\times 3$.



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.



6 Grafer

DEFINISJON 6.1. *En **graf** $\mathcal{G}$ består av en ikke-tom mengde noder $\mathcal{V}$ og en mengde kanter $\mathcal{E}$, slik at enhvet kant forbinder nøyaktig to noder med hverandre eller en node med seg selv.*

6.1 Terminologi

Vektet En graf er vektet dersom hver kant har en tredje komponent. En verdi langs kanten.

Sti En sti gjennom grafen er en sekvens av noder $v_1, v_2, v_3, \ldots, v_n$, slik at $(v_i, v_{i+1}) \in \mathcal{E}$ for $1 \leq i \leq n-1$.

Lengden til stien er lik antall kanter på stien.

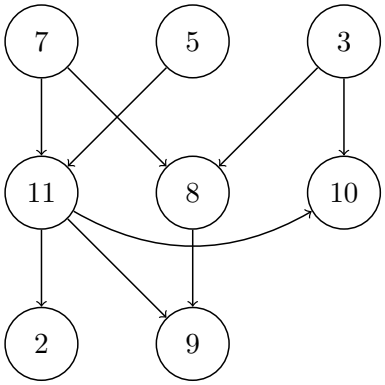
Løkke Ei løkke i en rettet graf er en vei med lengde ≥ 1, slik at $v_1 = v_n$. Løkken er *enkel* dersom stien er enkel.

Asyklisk En rettet graf er asyklisk dersom den ikke har noen løkker. DAG (Directed, Asyclic Graph).

Sammenhengende En urettet graf er sammenhengende dersom det fins en sti fra hver node til alle andre noder.

6.2 Topologisk sortering

I figur 5 har vi følgende lovlige topologiske sorteringer:



Figur 5: Eksempelgraf

- 7, 5, 3, 11, 8, 2, 9, 10 (visual left-to-right, top-to-bottom)
- 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
- 3, 7, 8, 5, 11, 10, 2, 9 (because we can)
- 5, 7, 3, 8, 11, 10, 9, 2 (fewest edges first)
- 7, 5, 11, 3, 10, 8, 9, 2 (largest-numered available vertex first)
- 7, 5, 11, 2, 3, 8, 9, 10 (attempting top-to-bottom, left-to-right)

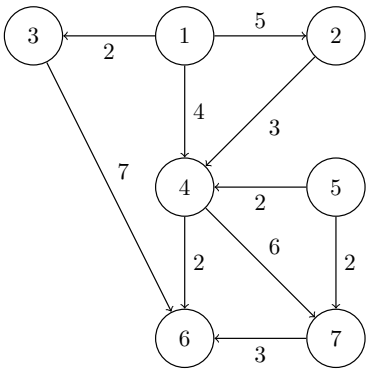
Algoritmer De vanligste algoritmene for topologisk sortering har lineær kjøretid i antall noder, pluss antall kanter; $O(|\mathcal{V}| + |\mathcal{E}|)$. En av disse kommer det et eksempel på her.

```
L <- empty list that will contain the sorted elements
S <- set of all nodes with no incoming edges

while S in non-empty do:
    remove a node n from S
    insert n into L
    for each node m with an edge e from n to m do:
        remove edge e from the graph
        if m had no other incoming edges then:
            insert m into S
if graph has edges then:
    return error (graph has at least one cycle)
else
    return L (a topofically sorted order)
```

6.3 Dijkstra

1. For alle noder, sett avstanden fra startnoden s lik ∞. Merk noden som *ukjent*.
2. Sett avstanden fra s til seg selv lik 0.
3. Velg en ukjent node v med minimal avstand fra s og marker v som kjent.
4. For hver ukjente nabonode w til v: Dersom avstanden vi får ved å følge veien gjennom v, er kortere enn den gamle avstanden til s. Redusér avstanden til s for w og sett bakoverpekeren i w til v.
5. Akkurat som for uvektede grafer, ser vi bare etter potensielle forbedringer for naboer som ennå ikke er kjent.



Figur 6: Eksempelgraf for Dijkstra.

Eksempel på Dijkstra: teorioppgave

	Init			**1**			**3**		
V	known	dist.	from	known	dist.	from	known	dist.	from
1	F	0	0	T	0	0	T	0	0
2	F	–	0	F	5	1	F	5	1
3	F	–	0	F	2	1	T	2	1
4	F	–	0	F	4	1	F	4	1
5	F	–	0	F	–	0	F	–	0
6	F	–	0	F	–	0	F	9	3
7	F	–	0	F	–	0	F	–	0

	6			**4**			**7**		
V	known	dist.	from	known	dist.	from	known	dist.	from
1	T	0	0	T	0	0	T	0	0
2	F	5	1	F	5	1	F	5	1
3	T	2	1	T	2	1	T	2	1
4	F	4	1	T	4	1	T	4	1
5	F	–	0	F	–	0	F	–	0
6	T	9	3	T	6	4	T	6	4
7	F	–	0	F	10	4	T	10	4

	2		
V	known	dist.	from
1	T	0	0
2	T	5	1
3	T	2	1
4	T	4	1
5	F	–	0
6	T	6	4
7	T	10	4

Tabell 3: Tabell til Dikjstra-algoritmen på grafen i figur 6.

Algoritmen Her er pseudokoden for Dijkstra.

```
function Dijkstra (Graph, source):
    for each vertex v in Graph:
        distance[v] := infinity
        visited[v] := false;
        previous[v] := undefined;

    distance[source] = 0;
    insert source into Q;

    while Q is not empty:
        u := vertex in Q with smallest distance in distance[]
                    and has not been visited;
        remove u from Q;
        visited[u] := true;

        for each neighbour v of u:
            alt := distance[u] + dist_between(u,v);
            if alt < dist[v];
                dist[v] := alt;
                previous[v] := u;
                if !visited[v]:
                    insert v into Q;
    return distance;
end function;
```

6.4 Prim

6.5 Kruskal

6.6 Dybde-først

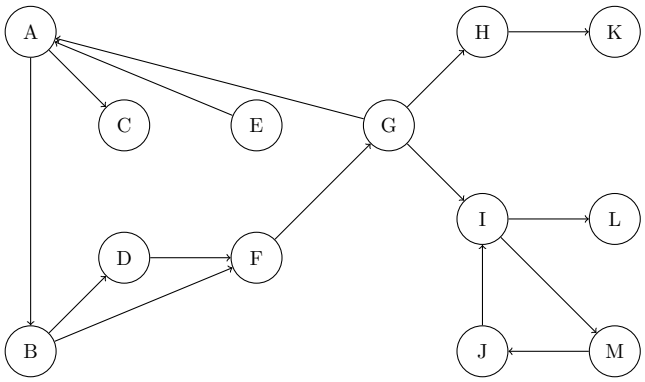
Dette er klassisk graf-traversering, generalisering av prefiks traversering for trær. Gitt start node v: rekursivt traverser alle nabonodene.

```
1 public void depthFirstSearch(Node v)
2   v.marked = true;
3   for <each neighbour w to v>
4     if (!w.marked)
```

```
5           depthFirstSearch(w);
```

6.7 Biconnectivity

DEFINISJON 6.7.1. *En sammenhengende urettet graf er* ***bi-connected*** *hvis det ikke er noen noder som ved fjerning gjør at grafen blir usammenhengende. Slike noder heter cut-vertices eller articulation points.*



Figur 7: Eksempelgraf for bi-connectivity.

Grafen i figur 7 er *ikke* bi-connected, fordi nodene A, I, G og H er articulation points.

6.8 Strongly connected components (SCC)

DEFINISJON 6.8.1. *Gitt en rettet graf $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. En* ***strongly connected component*** *av $\mathcal{G}$ er et maksimalt sett av noder $U \subseteq \mathcal{V}$. For alle $u_1, u_2 \in U$ har vi at $u \to^* u_2$ og $u_2 \to^* u_1$.*

7 Kombinatorisk søk

8 Rekursjon

9 Tekstalgoritmer

Tabeller

Figurer