Algoritmer og datastrukturer

Skrevet av:

Are Nybakk

Avgangsstudent *Bachelor Ingeniør Datateknikk* v/NITH 2007-2008

Innhold

Innledning

Som avgangsstudent på studieretningen *Bachelor Ingeniør Datateknikk* ved NITH ønsket jeg, som en del av eksamensforberedelsene, å gjengi pensumet for faget *Algoritmer og datastrukturer* i skriftlig form. Dette var både for meg selv og for eventuelle andre studenter. Algoritmer og datastrukturer var et av fagene som var del av pensumet i tredje klasse. Teksten er basert på de sentrale temaene og eksemplene som ble gjennomgått i forelesningene. Foreleser var Bjørn Krogdahl og pensumlitteraturen var boken *Data Structures and Algorithm Analysis in Java* skrevet av Mark Allen Weiss.

Faget gir en introduksjon til sentrale begreper innen algoritmeteori samt en del grunnlegende datastrukturer. Helt først skal vi se på litt enkel matematikk som gir et grunnlag for noe av teorien i resten av pensumet. Det kreves ikke spesielle forkunnskaper i matematikk annet enn enkel algebra og potensregning. Deretter skal vi rask se eksempler på såkalt *rekursjon*. Vi skal så se på datastrukturbegrepene *stack*, *tre*, *heap* og *hashing*. Deretter bringer pensum oss over på sorteringsalgoritmer. Avsluttningsvis skal vi se på såkalte *grafer* og *greedy-algoritmer*.

Jeg håper denne teksten kan hjelpe med forståelsen for stoffet for eventuelle andre lesere. Gi meg gjerne tilbakemelding på e-mail: *nith@arenybakk.com*.

1 Matematikkgrunnlag

1.1 Rekker og summer

En rekke er en følge av tall separert med en operator. Hvert tall kaller vi et ledd. For oss er det ikke nødvendig å gå inn på hvilke typer rekker vi har, men vi bør kjenne til hva det er for noe og hvordan vi kan beregne summen av dem. Under er en enkel rekke. Rekken er uendelig. Det vil si at følgen fortsetter og dette beskriver vi ved å sette inn tre prikker.

$$1 + 2 + 3 + \cdots$$

Summen av alle leddene angis med følgende notasjon:

$$S = \sum_{i=1}^{5} 2i = 2 + 4 + 6 + 8 + 10 = 30$$

Her er det spesifisert at summen er lik alle leddene addert sammen, der hvert ledd har formen *2i* og *i* er mellom 1 og 5. I programmeringssammenheng kunne denne summen beskrive hvor mange operasjoner en løkke innebærer. *2i* er da antall operasjoner for én iterasjon og *i* er iteratoren (tellevariabelen). En slik løkke kan være:

```
for(int i=1; i<=5; i++) {
    /*Operasjoner*/
}
```

For å finne et uttrykk for summen av en rekke kan vi bruke et triks. La oss se på en binær rekke som et eksempel. Vi multipliserer alle ledd med 2 og får dermed en likning nummer to. Vi har fått et likningssett. Ved å trekke likningene fra hverandre vil vi få en enkel likning da mange av leddene kan strykes mot hverandre. Det var derfor multiplikasjon med 2 ble valgt.

$$S = 2^0 + 2^1 + 2^2 + \cdots + 2^{n-1}$$
$$2S = 2^1 + 2^2 + 2^3 + \cdots + 2^n$$
$$2S - S = 2^n - 2^0$$
$$S = 2^n - 1$$

1.2 Funksjoner og vekst

Funksjoner er likninger som bestemmes av en eller flere inn-parametere. Notasjon for en funksjon *f* som beskriver likningen $3a + 2$, der *a* er et vilkårlig tall, er:

$$f(a) = 3a + 2$$

$$f(5) = 15 + 2 = 17$$

Om vi ser på resultatene når vi inkrementerer inn-parameteren kan vi si noe om veksten. Vi må kjenne til noen ulike grunntyper av vekst:

--sett inn lineære, kvadratiske og eksponentielle grafer---

I tillegg må vi kjenne til logaritmer. En logaritme er intet annet enn en invers, eller motsatt, funksjon. Om vi bruker den inverse funksjonen på resultatet vi får av den egentlige funksjonen, er vi tilbake til utgangspunktet. Den motsatte funksjonen til *f(a) = 3*a + 2* er:

$$a = \frac{f(a) - 2}{3}$$

$$a = \frac{17 - 2}{3} = 5$$

Vi snakker om logaritmer når vi har en funksjon på formen under. Et grunntall er representert med bokstaven *n* og kommer igjen i logaritme-notasjonen. Dersom *n* ikke er spesifisert er det underforstått at *n = 10*.

$$f(a) = n^a$$

$$a = \log_n(f(a))$$

--sett inn logaritmisk v.s. kvadratisk/lineær graf---

Som vi kan se har logaritmiske funksjoner en mindre dramatisk vekst enn for eksempel kvadratiske. Når vi skal skrive effektive algoritmer ønsker vi å oppnå en *kjøretid* (tiden det tar å utføre en oppgave) som følger en logaritmisk utvikling.

1.3 O-notasjon

Når vi skal skrive et program som skal behandle store mengder data er kjøretiden avgjørende. Et nyttig verktøy i slike situasjoner er såkalt *O-notasjon*. Den matematiske, teoretiske beskrivelsen er komplisert og vi skal kun se på hva det innebærer. Det hele er temmelig enkelt. Når vi snakker om store mengder data må vi først og fremst se på kjøretiden til det som tar lengst tid. Det er dette som er det essensielle. O-notasjonen er en forenkling der vi ser bort fra alle konstanter og ledd som er av lavere grad (potenser med lavere eksponent) enn høyeste forekommende grad. Følgende kjøretid T er en funksjon av datamengden *N* og O-notasjonen for den er angitt med *O( )*.

$$T(N) = 3N^2 + 2N = O(N^2)$$

1.4 Modulus

Moduloregning er egentlig ikke annet enn at vi sier noe om hvor mange siffer vi regner med. Hvis vi velger 5 siffer er mulige siffere 0, 1, 2, 3 og 4. Da er det slik at $4 + 1 = 0$ ettersom neste mulige tall er 0. Man kan tenke på sifferne som posisjoner på et bånd som er festet sammen i endene. Restdivisjonsoperatoren i programmeringsspråk gir oss samme resultat (*i % 5)*. Det finnes kun to regneoperasjoner; addisjon og multiplikasjon. Når vi ønsker å fjerne et siffer fra den ene siden av likningen adderer vi et tall slik at vi får 0. Så lenge vi gjør dette på begge sider er det lovlig. Når vi ønsker den ukjente alene ønsker vi å oppnå en koeffisient på 1 (*1x = x*) og dette gjør vi ved å multiplisere på begge sider. Under er utregningen for en likning med modulo 10 gitt. Hvis en ser etter er det egentlig akkurat dette vi gjør ved vanlig algebra. Resultatene blir nøyaktig de samme. At vi flytter siffere og skifter fortegn er en forenkling av sannheten.

$$3x + 1 = 6 \, mod \, 10$$

$$3x + 1 + 9 = 6 + 9 \, mod \, 10$$

$$3x + 0 = 5 \, mod \, 10$$

$$7(3x) = 7(5) \, mod \, 10$$

$$x = 5$$

Vi kan sette opp tabeller for regneoperasjonene. Her velger vi modulo 4:

+	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

*	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	0	2
3	0	3	2	1

Det viser seg at det er interessant å se på multiplikasjonstabellen. Om vi ser på multiplikasjon med 2 forekommer 0 to ganger. Når vi ønsker å generere et tall fra et annet kan vi ikke ha slik tvetydighet. En likning som illustrerer problemer er gitt under. Det finnes altså to korrekte svar. Dette er som regel ikke ønskelig. Man kan tenke seg en situasjon der datamaskinen skal plukke et stykke data men det finnes to som oppfyller kravene. Hva gjør maskinen?

$$2x = 0 \ mod \ 4$$

$$2(0) = 0 \ mod \ 4$$

$$2(2) = 0 \ mod \ 4$$

Det viser seg at dersom vi bruker primtall vil slik tvetydighet aldri kunne forekomme. Dessuten er sifferene godt spredt i tabellen. Derfor brukes primtall for krypteringer. Under er multiplikasjonstabellen for modulo 7 gitt. Hvis vi antar at vi i dette tilfelle har en krypteringsnøkkel som er 3 gjengis tallet 123 som 362. Ved dekryptering med riktig nøkkel vil 362 igjen gi 123. En uønsket person som ikke har riktig krypteringsnøkkel vil få et helt annet resultat. Dersom vedkommende har nøkkelen 5 vil 362 gi tallet 246. Dette er illustrert i tabellen. Moduloregning og primtall dukker opp når vi skal se på hashing.

*	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6
2	0	2	4	6	1	3	5
3	0	3	6	2	5	1	4
4	0	4	1	5	2	6	3
5	0	5	3	1	6	4	2
6	0	6	5	4	3	2	1

2 Rekursjon

Et nyttig verktøy i programmering er *rekursjon*. Rekursjon har vi når en funksjon kaller seg selv. Slike kalles *rekursive kall*. Tanken bak å bruke rekursjon er å gjøre et problem lettere ved å dele opp oppgaven i mindre oppgaver. Vi deler opp helt til vi har oppnådd en helt elementær oppgave – et såkalt *base case*. Dette resulterer i at man kan løse svært komplekse problemer med noen få linjer kode. En slik algoritme er en såklalt *divide and conquer*-algoritme. Vi skal se på et klassisk skoleeksempel som illustrerer dette.

2.1 Fibonachirekken

Fibonachirekken har den egenskapen at de første to tallene er 1 og de påfølgende er lik summen av de to foregående. Vi kan benytte rekursjon for å løse en slik oppgave og grunnen til det er at vi kan se to grunntilfeller og et fast gjentakende mønster. Mønsteret er at hvert tall er lik to andre tall. For siffer nummer n vet vi at dersom n=0 (første fosisjon er 0) eller n=1 er resultatet 1 (base case) og alle andre verdier n er lik summen av tallet for n-1 addert med tallet for n-2:

Fibonachirekken: 1 + 1 + 2 + 3 + 5 + 8 + 13 + 21 + ...

$$fib(0) = 1$$

$$fib(1) = 1$$

$$fib(n) = fib(n-1) + fib(n-2)$$

$$fib(2) = fib(1) + fib(0) = 1 + 1 = 2$$

$$fib(3) = fib(2) + fib(1) = 2 + 1 = 3$$

Hvis vi ser på fib( ) som en metode som returnerer summen av de n første leddene så kan vi raskt og enkelt sette opp koden for den:

```java
public int fib(int n) {
    if(n == 0 || n == 1) {
            return 1;
    }
    else {
        return fib(n-1) + fib(n-2);
    }
}
```

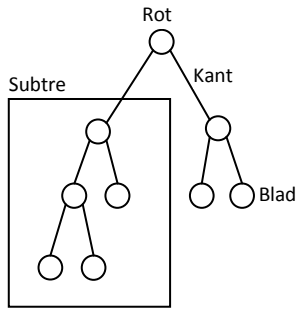
2.2 Stack

En stack er en lagringsmetode der siste element som blir lagt inn er første element som blir tatt ut (*FILO* – first In, last out). Man kan tenke på det som en stabel med tallerkener på en benk og man tar alltid den øverste tallerkenen ut. For en stack er det to operasjoner vi trenger. Vi trenger en operasjon for å legge til et element (*push*) og en for å ta ut (*pop*). Det er også vanlig med en operasjon for å se elementet som ligger øverst uten å ta det ut (*top*) og en for å se om stacken er tom.

Når det gjøres kall på metoder i et program benyttes det en stack. Variabler som er nye i en metode legges på en stack før kode blir kjørt og fjernes igjen når koden er utført. Det er viktig å ha i tankene at dersom man gjør mange rekursive kall vil minnet i maskinen sakte fylles opp fordi at metoden ikke kan fortsette før metoden den kalte er ferdig. Slik fortsetter kallene nedover inntil en metode har nådd et base case. Da returnerer denne slik at de forrige kan fortsette og returnere osv. (*nøste opp*). Det er et stort problem dersom et base case aldri nåes.

3 Trær

Trær er en struktur der elementer ligger hierarkisk lagret. Et element kalles en *node*. Et tre har en node øverst i hierarkiet som vi kaller en *rotnode*. En node kan ha ingen, en eller flere noder under seg (*barn*). En node som har barn kalles en *mor* mens en uten barn kalles et *blad (løv* i flertall). Et tre kan deles inn i flere trær (*subtrær*) der en node er subtreets rotnode. En forbindelse mellom noder kalles en *kant*. Vi skal se på binære trær. Disse har den egenskapen at hver node kan ha maks to barn – et venstre og et høyre barn. En nodes venstre barn utgjør roten for venstre subtre og likeledes for høyre barn. Hver node må ha ha en *nøkkel* (*key*) og referanser til sine barn.



3.1 Traversering

Traversering er å gå igjennom alle nodene i et tre og gjøre en operasjon på dem. Det finnes tre måter å gjøre dette på: *pre-order*, *in-order* og *post-order*. Forskjellen er rekkefølgen på operasjonene:

Pre-order	1) denne noden 2) venstre subtre 3) høyre subtre
In-order	1) venstre subtre 2) denne noden 3) høyre subtre
Post-order	1) venstre subtre 2) høyre subtre 3) denne noden

Dette lar seg løse rekursivt. Under er et eksempel på bruk av pre-order.

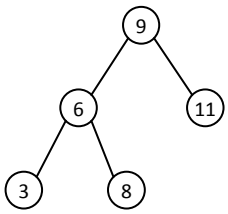
```
void traversePreorder( Node thisNode ) {

    /* Kun dersom referansen er gyldig (noden finnes) */
    if(thisNode != null) {
        thisNode.task();
        traversePreorder(thisNode.getLeftChild());
        traversePreorder(thisNode.getRightChild());
    }

}
```

3.2 Slette en node

I binære trær er det tre tilfeller som kan oppstå når vi ønsker å slette en node. Dersom noden er et blad trenger vi bare å fjerne referansen hos mornoden. Dersom noden har ett barn bytter vi referansen hos mornoden med en referanse til dette barnet. Dersom noden har to barn må vi lete i høyre subtre etter den noden som har minst nøkkel og referere den i mornoden. For sistnevnte må vi til slutt også passe på å legge eventuelle barn til noden vi flyttet, inn på riktig plass i treet.

3.3 Binært søketre

Et binært søketre er et binært tre der venstre og høyre subtre inneholder noder med henholdsvis lavere og høyere nøkler. Et lite eksempel på et gyldig binært søketre er gitt under. For å sette inn en node må man da gå nedover i treet og avgjøre om den hører hjemme i venstre eller høyre subtre for hvert trinn. Når vi finner en ledig plass setter vi noden inn.



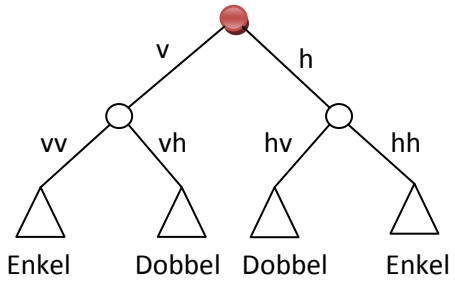
3.4 Balanserte trær - AVL-trær

Et ubalansert tre vil kunne få veldige skjevheter i fordelingen av noder mellom høyre og venstre subtrær (*vekt*) og effektiviteten vil dermed også bli dårlig. Et AVL-tre er et binært søketre der vi har satt følgende krav: forskjellen i antall nivåer mellom venstre og høyre subtrær må ikke overstige 1. Vi må innføre noen nye bregreper. *Dybden* til en node er lengden på *stien*. En sti er rute fra roten til

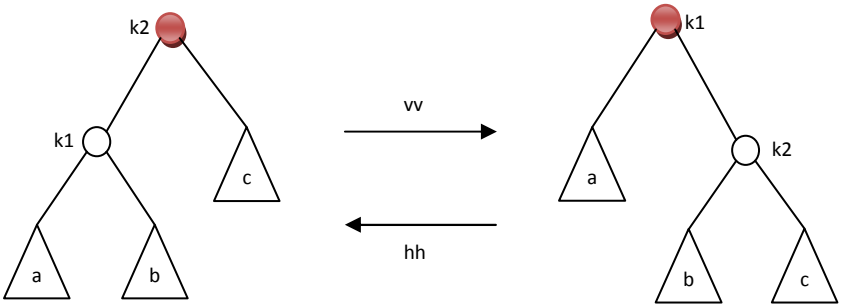
noden og lengden på den er antall kanter. *Høyden* til en node er den lengste stien fra seg selv til et blad. En node i et AVL-tre må holde rede på sin høyde.

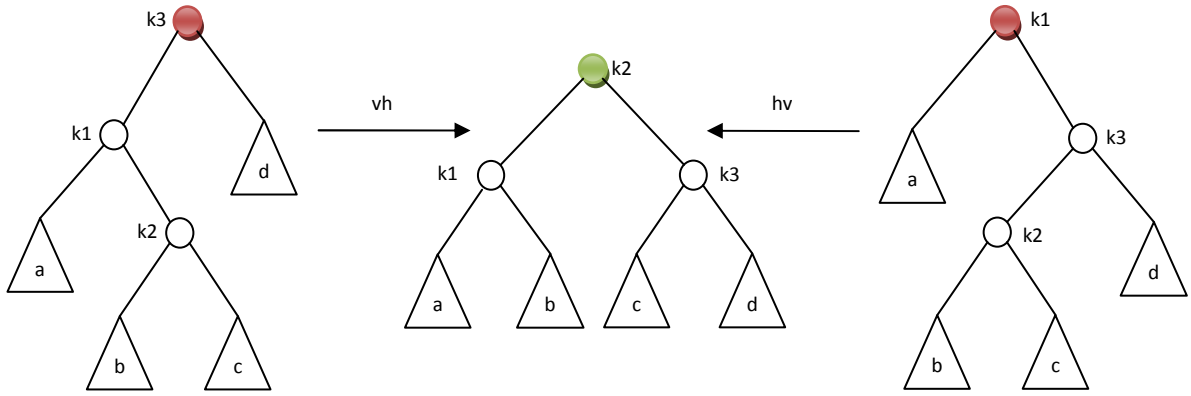
3.4.1 Balansering med rotasjoner

Når vi skal legge til et element gjør vi rekursive kall til vi finner en ledig plass. I oppnøstingen, opp til roten, må vi korrigere høyden i mornodene og sammenligne høydeforskjellen mellom venstre og høyre subtre. Når vi eventuelt finner en *problemnode* må vi gjøre en justering. Avhengig av hvor vekten ligger må vi foreta enten en enkel eller en dobbel rotasjon. Diagrammet under illustrerer hvilken type korrigering vi må foreta. Trekantene tilsvarer subtrær og det er subtreet som har størst høyde (veier mest) som avgjør type.



Grunnen til at man kaller operasjonene for rotasjoner er at man flytter om på nodene på en rullende måte. Rotasjonene er illustrert under. Det er litt vanskelig å se rotasjonsmønsteret i dobbel rotasjon, men navnet har den fått fordi den kan gjøres med to enkle rotasjoner. Dette skal vi se på etterpå. Nodene er gitt navn på formen kX der X understreker forhold i nøklene. K1 har da en lavere nøkkel enn k2 osv.



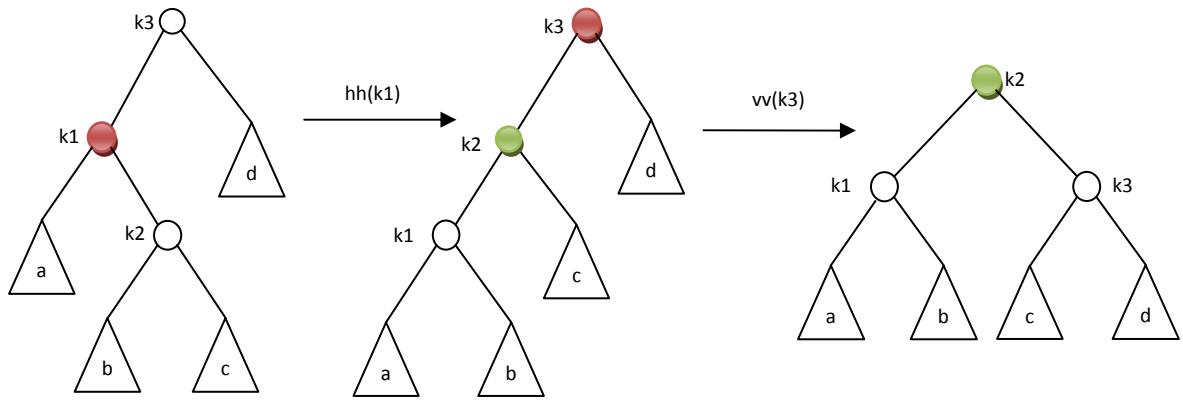


Vi trenger fire ulike metoder for å håndtere disse tilefellene. Under er eksempler på tilfellene vv og vh. De to andre vil være tilsvarende, bare speilvendt. Innparameteren er problemnoden og metoden returnerer noden som skal ta over plassen.

```
Node vv( Node k2 ) {

    Node k1 = k2.getLeftChild();
    k2.setLeftChild(k1.getRightChild());
    k1.setRightChild(k2);

    /* Juster høyder ved å finne høyeste barn og addér med 1 */
    k2.setHeight( Math.max(
            k2.getLeftChild().getHeight(),
            k2.getRightChild().getHeight()) + 1 );

    k1.setHeight( Math.max(
            k1.getLeftChild().getHeight(),
            k2.getHeight()) + 1 );

    return k1;

}
```

```
Node vh( Node k3 ) {

    Node k1 = k3.getLeftChild();
    Node k2 = k1.getRightChild();
    k1.setRightChild(k2.getLeftChild());
    k3.setLeftChild(k2.getRightChild());
    k2.setLeftChild(k1);
    k2.setRightChild(k3);

    /* Juster høyder */
    k1.setHeight( Math.max(
            k1.getLeftChild().getHeight(),
            k1.getRightChild().getHeight()) + 1);
    k3.setHeight( Math.max(
            k3.getLeftChild().getHeight(),
            k3.getRightChild().getHeight()) + 1);
    k2.setHeight( Math.max(
            k1.getHeight(),
            k3.getHeight()) + 1);

    return k2;

}
```

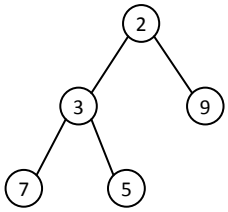
Som sagt er det mulig å gjøre en dobbel rotasjon med to enkle rotasjoner. Metodene for dobbel rotasjon blir da meget enkle. Dette er illustrert under.




```
Node vh( Node k3 ) {

    k3.setLeftChild(hh(k3.getLeftChild()));
    return vv(k3);

}
```

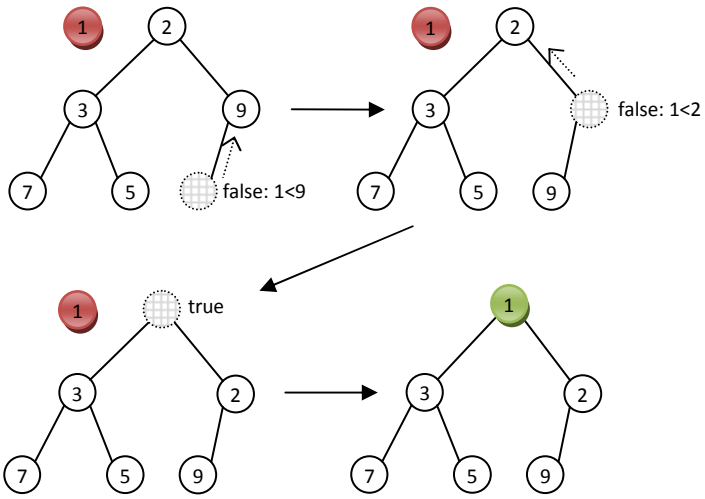
3.5 Heap

En *binær heap* er et såkalt *komplett binært tre*. Et binært tre er komplett når alle nivåer untatt det siste er fyllt opp og siste nivå er fylt opp fra venstre. Dette vil gjøre at treet alltid er balansert. Dessuten bygges en heap opp slik at mornoder alltid har en lavere nøkkel enn sine barn. Dette fører til at vi bestandig vil ha den minste nøkkelen som roten i treet. Av denne grunn kalles en heap også for en *prioritetskø*. Lavere nøkkel vil tilsvare høyere prioritet. Et eksempel på en gyldig heap er gitt under. Merk at en heap ikke er et binært søketre.



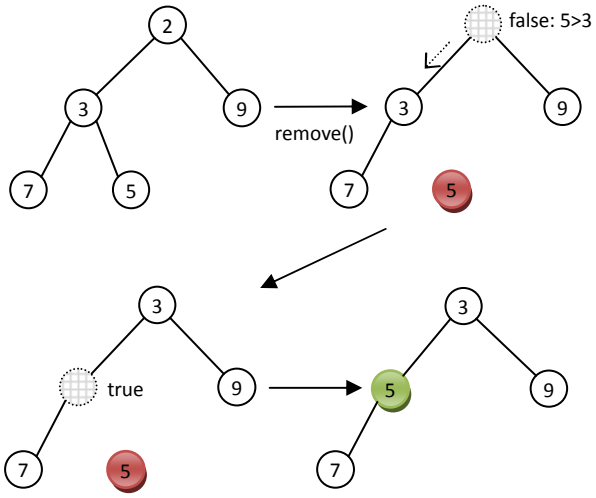
3.5.1 Legge til element

Vi må først finne neste gyldige posisjon i heapen. Posisjonen er gitt av kravene; den må være lengst mulig til venstre i siste nivå der det ikke finnes noen eksisterende node. I denne posisjonen plasserer vi et tenkt *hull*. Vi flytter dette hullet oppover, ved å bytte med moren, inntil vi finner en gyldig posisjon for elementet. Da kan vi erstatte hullet med elementet. Kravet sier at nøkkelen må være større enn morens nøkkel og mindre enn eventuelle barns nøkler. Denne prosessen kalles å *perkolere opp*. Et eksempel er gitt under.



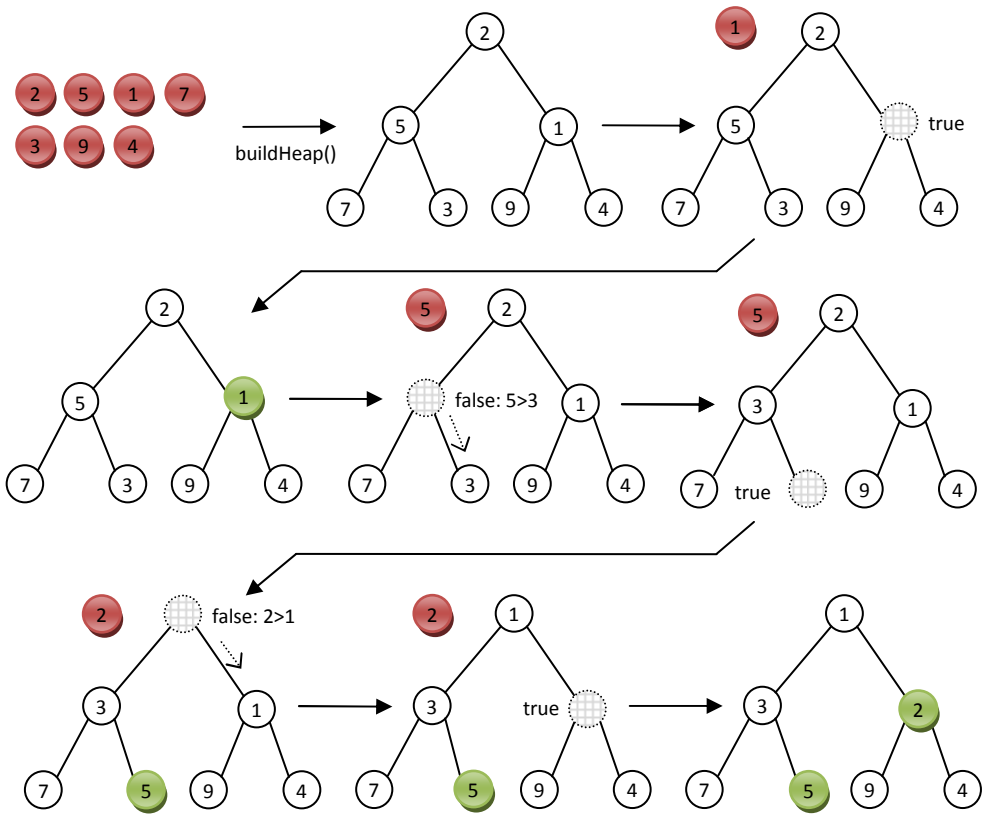
3.5.2 Ta ut element

Vi plukker ut elementet i roten og setter et hull i dets posisjon. Fordi at treet ikke er gyldig uten rot, må hente ut noden lengst til høyre i siste nivå og plassere det et sted i treet. Vi flytter hullet nedover ved å bytte det med det barnet som har minst nøkkel. Dette gjør vi inntil vi har en posisjon som tilfredstiller kravene. Da setter vi noden inn i denne posisjonen. Denne prosessen kalles å *perkolere ned*. Et eksempel er gitt under.



3.5.3 Buildheap

Buildheap er en metode som lar oss opprette en heap med et gitt datasett framfor å opprette et tomt og legge inn ett og ett element. Vi fyller inn dataene løpende i et komplett binært tre fra venstre til høyre. Når en rad er full fortsetter vi på neste. Nå fyller ikke treet nødvendigvis kravene til en heap slik at vi må rette det opp. Dette gjør vi ved å gå igjennom alle mornoder fra høyre mot venstre og oppover i treet. For hver mornode perkolerer vi ned slik at subtreet er korrekt. Når alle mornodene er gjennomgått slik, er treet en gyldig heap.



3.5.4 Array-implementasjon

En heap kan enkelt implementeres som en array. Vi legger nodene inn løpende fra venstre til høyre og nedover. For en node i posisjon *i* kan vi da finne dens venstre og høyre barn henholdsvis i posisjon 2*i og 2*i+1. Moren til en node finner vi ved å heltalsdividere posisjonen med 2. Noden med lavest nøkkel finner vi i posisjon 1 og siste node i siste fylte posisjon. Merk at posisjon 0 i arrayet ikke brukes. Treet vi nettopp bygget opp er gjengitt som et array under.

arraySize	...
size	4
	9
	5
	7
	2
	3
1	1
0	-

Når heapen vokser gjør arrayet også det. Det er ikke ønskelig å lage et nytt, større array hver gang et element legges til. Derfor setter vi en arraystørrelse som gir god plass. Da må vi ha en variabel som inneholder hvor mye av arrayet som er brukt. Dersom arrayet skulle bli fullt lager vi et nytt.

4 Hashing

Hashing går ut på å ha elementer i et array med en gitt størrelse slik at insetting, sletting og søk foregår med en omtrent fast hastighet. Elementets posisjon er bestemt av dets nøkkel. Vi må først ha en transformasjon, en såkalt *hash-funksjon*, fra nøkkelverdi til arrayposisjon. Modulusregning med tabellens størrelse gjør dette, men fordi arrayet normalt ikke vil ha nok posisjoner til alle mulige nøkler, vil det oppstå situasjoner da en posisjon allerede er opptatt. Vi trenger en måte å håndtere dette på. Det går for eksempel an å legge inn en lenket liste, men dette kan fort bli ineffektivt. Vi skal se på framgangsmåter for å finne en annen posisjon i arrayet. Vi ønsker å finne en slik funksjon som har stor spredning og er rask. Derfor velger vi også primtall for generering av nøkler i hashfunksjonen og som tabellstørrelse. Under står en funksjon for arrayposisjon for elementet med nøkkelverdi *x* der *i* er iteratoren. Hash-funksjonen er *hash(x)* og *f(i)* er en funksjon som bestemmer ny posisjon gitt *i*. For *f(i)* skal vi se på tre mulige funksjoner.

$$pos_i(x) = \big(hash(x) + f(i)\big) \bmod tableSize$$

4.1 Linear probing

Vi setter *f(i)* lik en lineær funksjon. Den enkelste lineære funksjonen er angitt under. Spredningen blir liten da neste posisjon alltid inkrementeres med et fast tall (1). Det oppstår lett klynger av data.

$$f(i) = i$$

4.2 Quadratic probing

Vi setter *f(i)* lik en kvadratisk funksjon. En slik er angitt under. Det viser seg at denne gir bedre spredning enn den lineære funksjonen, men at vi må sette noen krav for ikke å havne i en situasjon med evig løkke. Kravene er at arraystørrelsen må være et primtall og at lastfaktoren, λ, er mindre eller lik 0,5. Lastfaktoren er et tall mellom 0 og 1 (prosent) for hvor fullt arrayet er.

$$f(i) = i^2$$

4.3 Double hashing

Denne framgangsmåten bruker en ekstra hashingfunksjon. Vi skal ikke gå i detalj for Funksjonen *f(i)* er angitt under. *R* er et primtall mindre enn tabellstørrelsen.

$$f(i) = i * hash_2(x)$$

$$hash_2(x) = R - (x \bmod R)$$

4.4 Rehashing

Når arrayet blir fult må vi lage et nytt, større array. Vi kaller dette rehashing. For quadratic probing er dette noe vi gjør når lastfaktoren blir for stor. Vi velger da en ny arraystørrelse som er minst det dobbelte av den gamle. Deretter flytter vi dataene over.

5 Sorteringsalgoritmer

5.1 Mergesort

Mergesort er en av de to spesielle sorteringsalgoritmene vi skal se på. Oppgaven er å sortere et array. Mergesort går ut på å sortere venstre og høyre halvdel for seg. Problemet egner seg for rekursjon der base caset er å sortere ett element. Når halvdelene er sortert må vi *flette* (*merge*) de sammen til det endelige sorterte arrayet. Prinsippet bak flettingen er illustrert under. Merk at de to små arrayene egentlig er ett array delt i to. Det vi gjør er å sammenligne et element fra hver halvdel mot hverandre og skriver den minste til arrayet. Vi inkrementerer posisjonen i halvdelen vi plukket fra og fortsetter. Mergesort har ulempen at det kreves dobbelt så mye minneplass da det må opprettes et eget sorteringsarray.

Venstre sortert

i	
size/2	8
1	7
0	3

Høyre sortert

j	
size/2	6
1	5
0	2

i	j	minst	inkrementer
0	0	høyre	j
0	1	venstre	i
1	1	høyre	j
1	2	Høyre	- (j == size/2)
1	-	venstre	i
2	-	Venstre	- (i == size/2)

	Resultat
0	2
	3
	5
	6
	7
size	8

5.1.1 Kjøretid

Vi skal beregne kjøretiden for mergesort. Vi antar at arrayet er delelig med 2. Det betyr at antall elementer $N = 2^k$. Algoritmen har tre oppgaver; sortere venstre og høyre halvdel samt flettingen. Antall elementer for hver av halvdelene er halvparten av *N*. Vi gjør en tilnærming ved å si at flettingen tar *N* antall operasjoner da det kjøres i en løkke. Kjøretiden for en løkke er tilnærmet lik antall elementer da enkeltoperasjoner ikke har noe betydelig utslag. Vi kan sette opp en såkalt

rekursjonsformel som vist under. Vi skal se på to framgangsmåter for å finne kjøretiden gitt rekursjonsformelen.

$$N = 2^k$$

$$T(N) = T\left(\frac{N}{2}\right) + T\left(\frac{N}{2}\right) + N = 2T\left(\frac{N}{2}\right) + N$$

5.1.1.1 Framgangsmåte 1

Rekursjonsformelen vår inneholder kjøretiden for halve datamengden to ganger. To fordi vi har to rekursive kall. Vi finner et uttrykk for tiden et slikt rekursivt kall tar ved å sette $N = \frac{N}{2}$ i rekursjonsformelen. Nå kan vi sette opp uttrykket for den tiden 2 rekursive kall tar. Slik fortsetter vi et par ganger.

Ingen rekursive kall:

$$T(N) = 2T\left(\frac{N}{2}\right) + N$$

2 rekursive kall:

$$T\left(\frac{N}{2}\right) = 2T\left(\frac{N}{4}\right) + \frac{N}{2}$$

$$T(N) = 2\left(2T\left(\frac{N}{4}\right) + \frac{N}{2}\right) + N = 4T\left(\frac{N}{4}\right) + 2N$$

4 rekursive kall:

$$T\left(\frac{N}{4}\right) = 2T\left(\frac{N}{8}\right) + \frac{N}{4}$$

$$T(N) = 4\left(2T\left(\frac{N}{8}\right) + \frac{N}{4}\right) + 2N = 8T\left(\frac{N}{8}\right) + 3N$$

Vi kan se et mønster i utviklingen for hvert par med rekursive kall. Rekken er gitt under. Vi ønsker en funksjon for kjøretiden gitt av antall elementer *N*. Vi har sagt at $N = 2^k$ og dermed er $k = \log_2 N$. Det er selvfølgelig at $T(1) = 1$ (base case). Som vi ser under er kjøretiden for mergesort logaritmisk.

$$T(N) = 2T\left(\frac{N}{2}\right) + 4T\left(\frac{N}{4}\right) + 8T\left(\frac{N}{8}\right) + \cdots + N + 2N + 3N + \cdots = \sum_{k=1}^{\infty} 2^k T\left(\frac{N}{2^k}\right) + kN$$

$$T(N) = N + N\log_2 N = O(N\log_2 N)$$

5.1.1.2 Framgangsmåte 2 (teleskopering)

I rekursjonsformelen deler vi alle ledd med *N* fordi vi ønsker å få uttrykkene på begge sider på samme form. Vi ser så på alle tilefellene av *N* inntil *N=2* fordi vi da på høyresiden av likningen får *T(1)* (base case). Dette er vist under. Vi ser at vi kan stryke ut nesten alle ledd. Konstantleddene erstatter vi med *k*. Vi får samme resultat som med framgangsmåte 1.

$$\frac{T(N)}{N} = \frac{T\left(\frac{N}{2}\right)}{\frac{N}{2}} + 1$$

$$\frac{T\left(\frac{N}{2}\right)}{\frac{N}{2}} = \frac{T\left(\frac{N}{4}\right)}{\frac{N}{4}} + 1$$

$$\cdots$$

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

$$\frac{T(N)}{N} + \frac{T\left(\frac{N}{2}\right)}{\frac{N}{2}} + \cdots + \frac{T(2)}{2} = \frac{T\left(\frac{N}{2}\right)}{\frac{N}{2}} + \frac{T\left(\frac{N}{4}\right)}{\frac{N}{4}} + \cdots + \frac{T(1)}{1} + 1 + 1 + \cdots$$

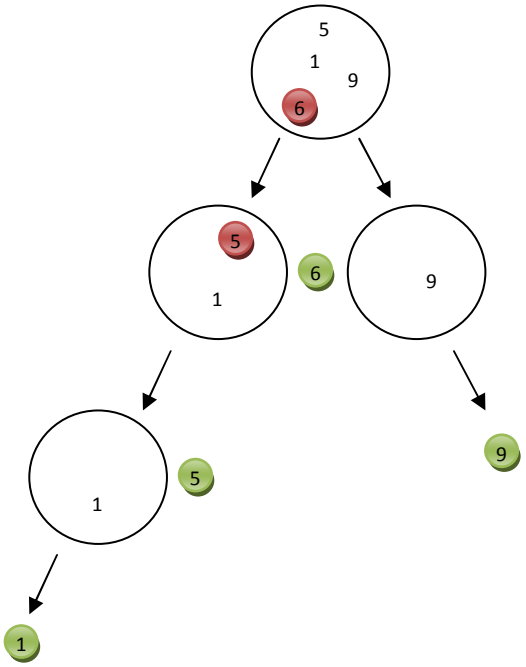
$$\frac{T(N)}{N} = T(1) + k = 1 + k$$

$$T(N) = N + kN = N + N\log_2 N = O(N\log_2 N)$$

5.2 Quicksort

Sorteringsalgoritmen quicksort deler også opp datasamlingen i to, men ikke på så enkelt vis som å dele et array på midten. Et såkalt *pivot-element* blandt dataene blir trukket ut. Alle data med nøkkel mindre enn pivot-elementet blir plassert for seg og det samme med de som har høyere nøkkel. Dersom flere elementer har samme nøkkel må man bestemme en måte å avgjøre fordelingen på.

Igjen benytter vi rekursjon. Base caset er en datasamling med ett eller ingen elementer. Dersom basecaset ikke er nådd må vi plukke et pivot-element og trekke det fra samlingen. Dette kalles *partisjonering*. Pivot-elementet vårt er klart for å plasseres. Det gjenstår å sortere de to samlingene. En enkel illustrasjon er gitt under.



5.2.1 Plukking av pivot-element

Det finnes flere tenkelige måter å plukke ut et slikt element på. Fordi det er en viss sannsynlighet for at et array allerede er delvis sortert unngår vi å bruke det første elementet. En annen mulighet ville være å trekke et tilfeldig element, men dette kan koste litt tid. Det vanlige er å benytte *medianen*. Medianen er rett og slett det midterste elementet. Det er strengt tatt ikke den ekte medianen som man snakker om i statistikkfaget, ettersom medianen er en verdi midt mellom de to midterste verdiene dersom det foreligger et oddetall med verdier. En slik mellomverdi vil ikke gi noen mening i dette tilfellet slik at vi må bestemme hvilket av de to midterste som skal benyttes.

5.2.2 Partisjonering

En måte å partisjonere på er som følger. Vi bytter pivot-elementet med siste element i arrayet. Vi setter så en iterator *i* lik posisjonen til første element og en annen iterator *j* lik posisjonen til nest siste element. Vi inkrementerer *i* inntil vi finner en nøkkel større enn pivot-elementet. Deretter dekrementerer vi *j* inntil vi finner en nøkkel mindre enn pivot-elementet. Når så begge iteratorene er stoppet bytter vi disse to elementene. Dersom *i* og *j* har krysset hverandre (*i* > *j*) skal pivot-elementet byttes med elementet *i* peker på. Dette resulterer i at elementene før pivot-elementet har mindre nøkler og elementene etter har større nøkkel. Et eksempel er gitt under.

3	7	1	4 (pivot)	0
0				size

3	7	1	0	4
i			j	

3	7	1	0	4
	i (7 > 4)		j (0 < 4)	

3	0	1	7	4
	i		j	

3	0	1	7	4
		j (1 < 4)	i (7 > 4)	

3	0	1	7	4
		j	i > j	

3	0	1	4	7
		j	i	

Dersom nøkler kan forekomme flere ganger får vi et dilemma. Iteratorene kan peke på element som har samme nøkkel som pivot-elementet. Det viser seg at det lønner seg å la både *i* og *j* stoppe dersom de støter borti dette.

5.2.3 Kjøretid

Algoritmen består i hovedsak av en løkke for partisjoneringen og to rekursive kall. Partisjoneringen tar N tid. Tiden for det første rekursive kallet er bestemt av antall elementer i det, *M*. Da er tiden for det andre rekursive kallet bestemt av *N-M-1* elementer da pivotelementet er blitt plassert (ett element). Rekursjonsformelen er da altså $T(N) = N + T(M) + T(N - M - 1)$. Vi skal se på to ekstremtilfeller og et gjennomsnittstilfelle.

5.2.3.1 Worst case

Det mest uheldige tilefellet er når delingen er 100% skjevt fordelt. D.v.s. at datasamlingen $M = 0$ og $T(M) = 0$. Da må vi igjennom hele arrayet og blir bare kvitt ett element av gangen, nemlig pivot-elementet. Rekursjonsformelen blir dermed $T(N) = T(N - 1) + N$. Vi benytter oss av teleskopering som vist under. Her er begge sidene på tilsvarende form.

$$T(N) = T(N - 1) + N$$

$$T(N - 1) = T(N - 2) + N - 1$$

$$T(N - 2) = T(N - 3) + N - 2$$

...

$$T(2) = T(1) + 2 = 1 + 2$$

Vi snur på rekken, noe som er helt lovlig. Vi summerer de to rekkene for å finne det generelle uttrykket for leddene. Kjøretiden blir $O(N^2)$ som tilsvarer en dobbelt løkke.

$$T(N) = 1 + 2 + \cdots + (N-2) + (N-1) + N$$

$$T(N) = N + (N-1) + (N-2) + \cdots + 2 + 1$$

$$2T(N) = N(1+N)$$

$$T(N) = \frac{N(N+1)}{2} = \frac{1}{2}N^2 + \frac{1}{2}N = O(N^2)$$

5.2.3.2 Best case

Det beste tilefelle som kan oppstå er at begge datasamlingene er omtrent like store. D.v.s. at dataene er delt på midten og begge rekursive kall tar like lang tid. Rekursjonsformelen blir da som under. Legg merke til at formelen i dette tilefelle blir lik den for mergesort. For gjentakelsens skyld teleskoperer vi og kommer fram til at kjøretiden blir logaritmisk.

$$T(N) = T\left(\frac{N}{2}\right) + T\left(\frac{N}{2}\right) + N = 2T\left(\frac{N}{2}\right) + N$$

$$\frac{T(N)}{N} = \frac{T\left(\frac{N}{2}\right)}{\frac{N}{2}} + 1$$

$$\frac{T\left(\frac{N}{2}\right)}{\frac{N}{2}} = \frac{T\left(\frac{N}{4}\right)}{\frac{N}{4}} + 1$$

...

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

$$\frac{T(N)}{N} = T(1) + k = 1 + k$$

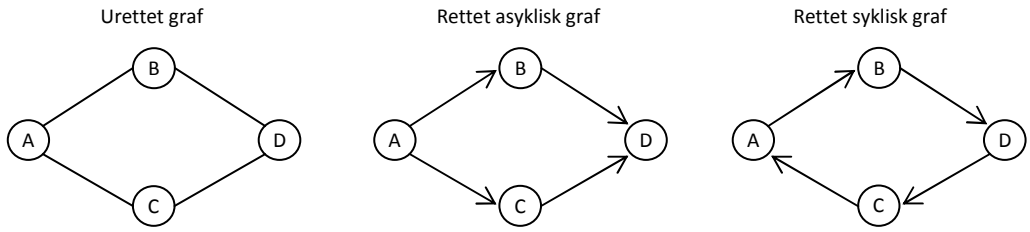
$$T(N) = N + N\log_2 N = O(N\log_2 N)$$

5.2.3.3 Average case

Vi skal ikke se på utregningen for gjennomsnittstilfellet, men det viser seg at kjøretiden også da blir logaritmisk: $T(N) = O(N\log_2 N)$.

6 Grafer og greedy-algoritmer

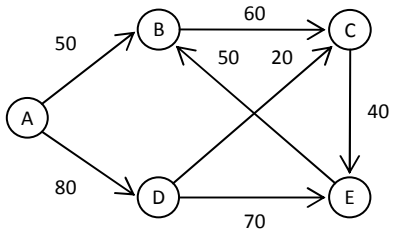
En graf er en måte å framstille noder og deres relasjoner med et fokus på sistnevnte. Relasjonene kalles, som med trær, for *kanter* mens nodene kalles *hjørner*. I en rettet graf har kantene en retning og i en urettetgraf er de uten retning. Dersom det i en rettet graf finnes en eller flere stier fra et hjørne tilbake til seg selv har vi en syklisk graf. Ellers har vi en asyklisk graf. En rettet asyklisk graf betegnes ofte med forkortelsen DAG.



Ensåkalt *greedy*-algoritme går alltid etter størst mulig gevinst ved hvert trinn. Et enkelt eksempel er en myntautomat som alltid gir ut høyest mulig valør når den skal utbetale en mynt.

6.1 Dijkstras algoritme

Dijkstras algoritme er en greedy-algoritme som benyttes for å finne den *minst kostbare* veien gjennom kjente hjørner. En rettet graf med kostnader gitt for hver kant er gitt under. Vi ønsker å sette opp en tabell som forteller hvor stor kostnad som er knyttet til andre kanter og hvilken sti som må tas.



Vi setter opp en kolonne for hver kant. På første rad fyller vi ut kostnadene fra hjørne *A*. Stien til disse er via A, derfor er A skrevet i parantes. De hjørnene som har en ukjent sti skrives opp med ∞. Vi

merker oppføringen i kolonnen for *A* med en * fordi den nå er kjent/har vært besøkt. Siden ser vi etter den minst kostbare ruten fra A som ikke er kjent. Vi ser nærmere på dette hjørnet og ser hvilke andre hjørner som kan nåes fra denne. Disse føres opp på samme vis som før, men dersom en oppføring allerede eksisterer skal den laveste kostnaden settes. Hjørner merket * skal ignoreres. Merk at kostnaden vi fyller inn alltid gjelder fra *A*. Når alle kantene er kjent har vi en tabell over alle de rimeligste rutene.

Gjeldende hjørne	A	B	C	D	E	Forklaring
A	0*	50(A)	∞	80(A)	∞	B: 50 < ∞ D: 80 < ∞
B	0*	50(A)*	110(B)	80(A)	∞	C: 50+60 < ∞ D: ∞ > 80
D	0*	50(A)*	100(D)	80(A)*	150(D)	C: 80+20 < 110 E: 80+70 < ∞
C	0*	50(A)*	100(D)*	80(A)*	140(C)	E:80+20+40 (C via D) < 150
E	0*	50(A)*	100(D)*	80(A)*	140(C)*	

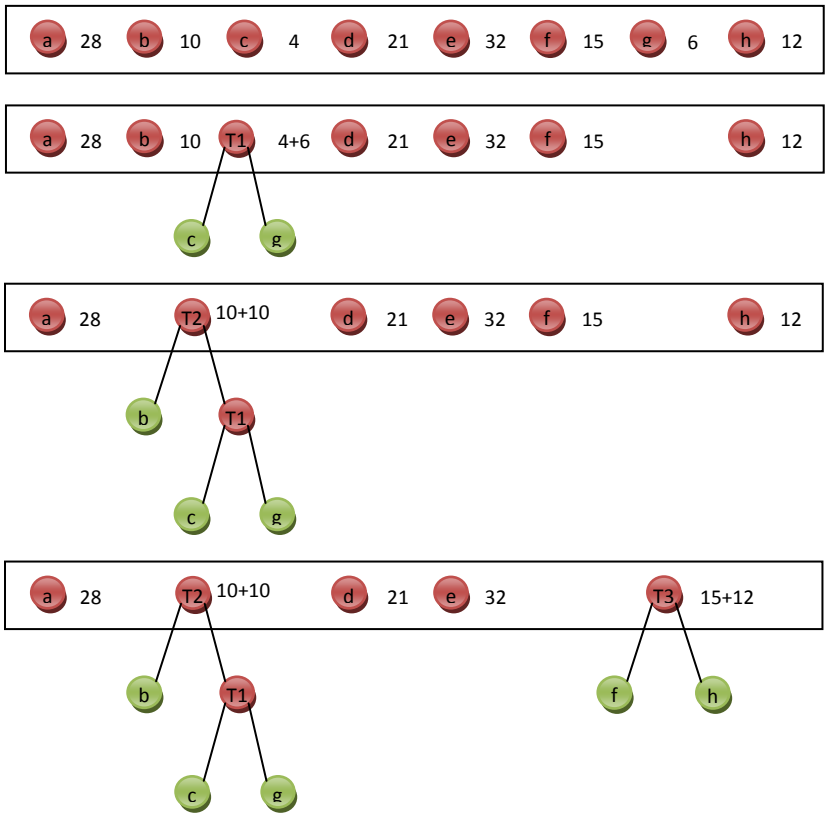
For å finne stien fra *A* til *E* ser vi på oppføringen i kolonnen for *E*. Den stien med minst kostnad koster 140. Vi ser på stien fra slutt til start. Første del av stien er *E fra C*. Vi slår opp for *C* og ser at rimeligste vei til *C* er fra *D*. Altså har vi litt mer av stien: *E fra C fra D*. Vi fortsetter slik og ser at stien blir *E fra C fra D fra A* eller *fra A til D til C til E*.

6.2 Huffmankoding

Huffmankode-algoritmen går ut på å representere mest forekommende data på mest ressursgjerrig måte. Vi skal se på hvordan vi kan komme fra til en optimalisering for tegnene *a-h* (8 tegn). Opprinnelig er hvert tegn representert binært med 3 bit ($2^3 = 8$). Ressursbruken vil da være fast uansett tegn og det er greit sålenge alle tegn har lik sannsynlighet til å forekomme (*uniform fordeling*). Dette er som regel ikke tilfelle i virkeligheten. Enkelte tegn forekommer mye oftere enn andre. For å finne en optimalisering trenger vi derfor data om hvor ofte de forekommer som vi legger inn i en tabell (*frekvenstabell*). Under er et eksempel på en slik.

tegn	a	b	c	d	e	f	g	h
frekvens	28	10	4	21	32	15	6	12

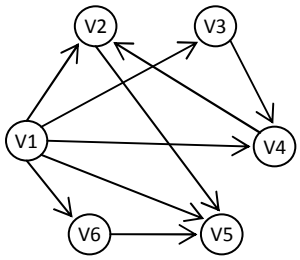
Vi setter opp et binært tre for den optimale koden. Alle venstre og høyre barn har henholdsvis binærverdiene *0* og *1*. Stien gir dermed koden for aktuelt tegn. Et slikt tre kalles et *optimalt binært kodetre*. Vi bygger opp treet ved å finne de to elementene som har lavest frekvens og knytte de til en node som barn. Dette gjør vi rekursivt intill kun ett element gjenstår (base case). Deler av oppbyggingen av treet for frekvenstabellen er vist under.



Osv.

6.3 Topologisk sortert liste

Dersom vi har en asyklisk rettet graf der det er maks en kant mellom to hjørner kan vi sortere dem etter avhengigheter. Dersom et hjørne har en rettet kant mot seg fra hjørne *X*, er denne avhengig av av hjørne *X*. Vi kan sortere en uoversiktlig graf slik at det forteller oss mer om forløpet. Dette kan være nyttig i enkelte situasjoner. Et godt eksempel er et prosjekt som er bygget opp i deler der disse delene er avhengige av at andre deler er fullført. Måten vi sorterer en slik graf på er å gå igjennom alle hjørnene og se hvilke som ikke har noen avhengigheter. Disse fjerner vi så ved å dekrementere antall avhengigheter (*inngrad*) hos de andre hjørnene denne var en forutsetning for. Dersom flere hjørner er blitt uavhengige legges de i en kø. Vi setter opp en tabell over alle hjørnene og hvor mange avhengigheter de har. Under er en usortert graf og framgangsmåten for sorteringen gitt.



V1	0
V2	2
V3	1
V4	2
V5	3
V6	1

V2	1
V3	0
V4	1
V5	2
V6	0

V2	1
V4	0
V5	2
V6	0

V2	1
V4	0
V5	1

V2	0
V5	1

V5	0

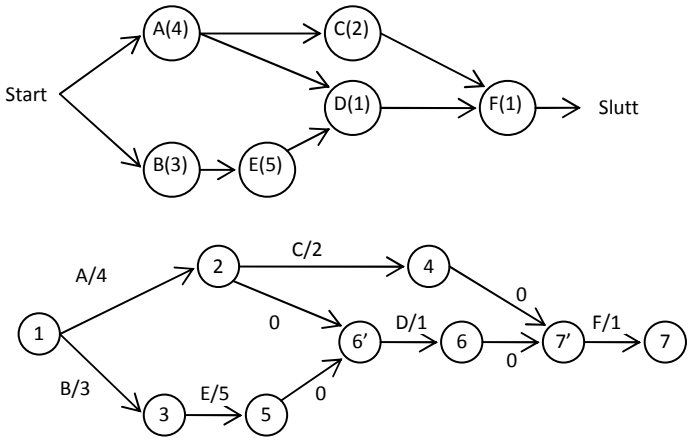
Res: V1, V3, V6, V4, V2, V5

6.4 Fullføringstid og kritiske stier

Vi vender tilbake til prosjekter som bruksområde. Vi vet hvor lang tid deloppgavene minst må ta og vi ønsker å finne ut hvor lang tid hele prosjektet må ta og hvilke deloppgaver som er kritiske. Andre deloppgaver kan vise seg å ha større tidsrom uten at det går ut over fullføringstiden for hele prosjektet. Dette kan benyttes som et verktøy for å finne ut hvor man bør fokusere ressursene i et prosjekt.

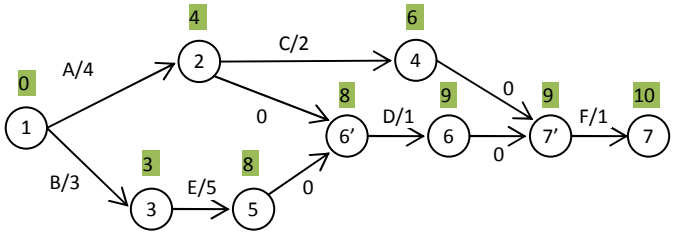
6.4.1 Aktivitetsgraf til hendelsesgraf

Vi har gitt en *aktivitetsgraf* der hjørnene er deloppgavene og tidsbruk er gitt i paranteser. En slik graf er gitt under. Vi ønsker å gjøre om denne grafen til en såkalt *hendelsesgraf*. En hendelsesgraf fokuserer på hendelser og disse hendelsene blir nummerert. Hendelsene tilsvarer at en aktivitet er fullført. Kantene fra en hendelse til en annen blir beskrevet med navn på aktivitet og tidsbruk. Aktiviteter som er avhengige av to andre blir delt opp i to hendelser. Den første, som kalles en *dummy-node*, representerer hendelsen der disse aktivitetene er fullført og den andre representerer aktiviteten selv. En dummy-node har ingen tidsbruk. Grafen er gjort om under. En dummy node har samme nummer som aktiviteten den er utsprunget fra men med en ′ (les: *merket*).



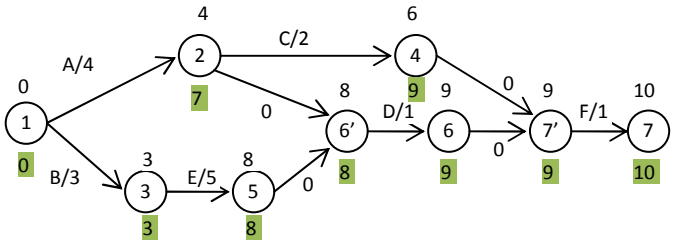
6.4.2 Tidligst fullføring

Tidligst fullføring av prosjektet finner vi om vi ser på lengste mulig sti fra start til slutt. Fra eksempelet kan vi se at aktivitet *A* og *E* må være ferdig før *D* kan starte. Altså er starttidspunkt for *D* avhengig av den lengste tiden av enten *A* eller *E*. Vi skriver tidsbruken ved hver tilstand (over). Dette er gjort under. Tidligst fullføring for dette prosjektet er 10.



6.4.3 Kritisk sti og slack

Når vi har funnet hvor lang tid prosjektet minst må ta kan vi studere nøyere hvilke aktiviteter som er kritiske. D.v.s. hvilke aktiviteter som ikke kan ta lenger tid enn planlagt uten at den totale tiden øker. Den totale tiden har vi funnet ut er 10. Vi baklengs fra siste hendelse til den første og trekker fra tiden til aktivitetene. Der to hendelser kommer fra én hendelse finner vi den minste. Et eksempel på dette er der tiden før hendelse 4 er 7 (9-2) og hendelse 6' er 8. Vi skriver tidsbruken ved hver tilstand (under). Resultatene er gitt under.



Om vi sammenligner tallene for tidligst fullføring med de vi fant nå, ser vi at det finnes avvik ved hendelse 2 og 4. Dette tolkes som følger. Aktivitetene før hendelse 2 kan ta opptil tiden 7, eller før hendelse 4 kan ta opptil tiden 9, uten at prosjektets tid økes. Dette omtales som *slack* (*slingringsmonn*). Hendelse 2 har et slingringsmonn på 3 (7-4) og 4 et på 3 (9-6). Den stien som ikke har noe slack fra start til slutt kalles en *kritisk sti*. Her er det ikke rom for utvidet tid om totaltiden skal holdes. Den kritiske stien er fremhevet under.