TOD063 Datastrukturer og algoritmer

Øving	: 4
Utlevert	: Veke 9
Innleveringsfrist	: 19. mars 2010
Klasse	: 1 Data og 1 Informasjonsteknologi

Ta gjerne 1 og 2 først!
Gruppearbeid: 2 personar pr. gruppe som tidlegare.

Oppgåve 1

a) Definer ein klasse: class Person implements Comparable<Person>{…}
med nødvendige metodar (konstruktør, hent-/sett-, toString- etc) slik at vi kan leggja objekt av av type Person inn i ein kø eller inn i ei ordna liste. La toString() returnera data i objektet på formatet:

```
fødselsår    etternamn, fornamn
```

Data i kvart objekt: - fornamn (String)
- etternamn (String)
- fødselsår (int)

For at du skal kunna bruka dei ferdige klassane for ordna liste må Person-klassen også innehalda ein metode compareTo(Person denAndre) som samanliknar objektet med parameteren den Andre, finn ut kven av dei som kjem først i ordna rekkefølgje, og returnerer med -1, 0 eller +1. Saman-likninga skal skje slik at yngre personar kjem før eldre. (compareTo() er metode i interface-et **Comparable**). Ved like fødselsår skal du samanlikna etternamn og deretter eventuelt fornamn (stigande alfabetisk rekkefølgje).

b) Lag eit enkelt main()-program der du opprettar ein *kø* av Person-objekt, les inn 4 – 6 objekt frå tastaturet og legg kvart objekt etter tur inn i køen. Til slutt tar du ut 1 og 1 objekt frå køen og skriv ut på skjermen inntil køen er tom. Bestem sjølv om du vil bruka Tabellkø2-klassen eller Kjedakø-klassen.

c) Lag eit enkelt main()-program der du opprettar ei *ordna liste* av Person-objekt, les inn 4 – 6 objekt frå tastaturet og legg kvart objekt etter tur inn i lista. Til slutt tar du ut 1 og 1 objekt frå lista og skriv ut på skjermen (stigande alder) inntil lista er tom. Pass på at du har fleire personar med same fødselsår. Bestem sjølv om du vil bruka TabellOrdnaListe-klassen eller KjedaOrdnaListe-klassen.

Krav til dokumentasjon av oppgåve 1 ved innlevering:

Programlisting av Person-klassen og dei 2 main()-programma. Utskrift av kjøring i b) og c).

Oppgåve 2

Innledning

Begrep: En *jobb* eller *prosess* er et program under utføring.

Et *operativsystem* er det overordnete programmet i datamaskinen som bl.a. kontrollerer prosesser og tildeler ressurser til disse. Operativsystemet har en rekke arbeidsoppgaver hvorav en er tildeling av CPU-tid (kjøretid). Algoritmen for en slik oppgave kalles *kjøreplanalgoritme*.

Den enkleste algoritmen er å utføre alle jobbene ferdig etter tur, dvs. vanlig kø-ordning. Men med en slik strategi risikerer vi at en jobb med lang kjøretid beslaglegger CPU-en for en lengre periode slik at korte jobber får urimelig lang ventetid.

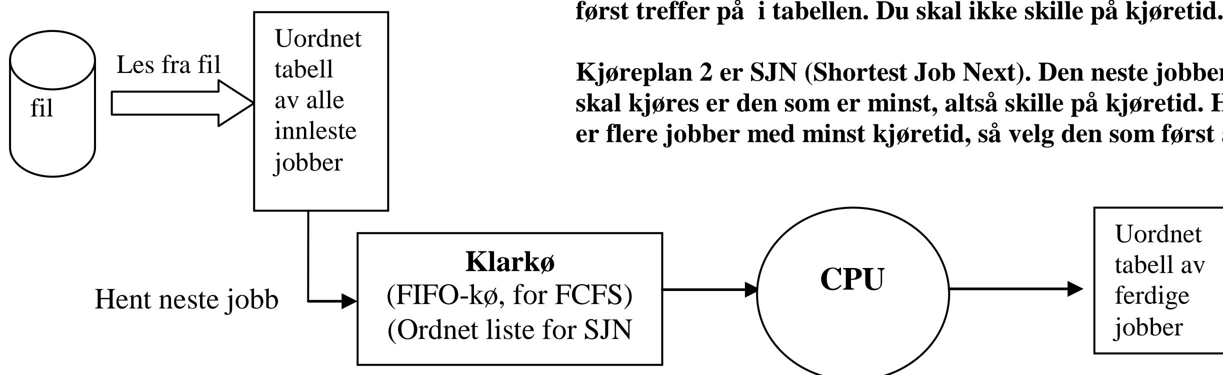
Et alternativ er å alltid velge ut den korteste jobben først når CPU er ledig for neste oppgave, dvs. at jobbene ligger og venter i en ordnet liste, med de korteste først. Dermed får små jobber kort svartid.

(Reelle operativsystemer bruker ofte en kombinasjon av disse teknikkene, men og det kan dere teste ut i en frivillig tilleggsoppgave 2 b).

Skisse av systemet:

Kjøreplan 1 er FCFS (First Come First Served). Den neste jobben som skal kjøres er den som ankom først. Hvis det er flere jobber som har ankommet først, skal du ta den som du først treffer på i tabellen. Du skal ikke skille på kjøretid.

Kjøreplan 2 er SJN (Shortest Job Next). Den neste jobben som skal kjøres er den som er minst, altså skille på kjøretid. Hvis det er flere jobber med minst kjøretid, så velg den som først ankom.



Vi studerer en modell for en kjøreplanalgoritme med *en* prosessor (CPU) og en mengde av jobber som er klar for utføring. Denne klar-køen er i første forsøk en *kø* (f. eks. Tabellkø2 eller KjedaKø), og i andre forsøk en *ordnet liste* (f.eks. KjedaOrdnaListe eller TabellOrdnetListe) der jobber med *kortest kjøretid* står først i listen (ved like kjøretider sammenligner vi ankomsttider).

Når CPU-en er ferdig med en jobb sjekker den om nye jobber har ankommet. I så fall blir disse lagt inn i køen/den ordnete listen. Deretter velger den ut neste jobb for utføring.

For å vurdere hvor god en kjøreplanalgoritme er, er det vanlig å se på gjennomsnittlig ventetid for jobbene i systemet i løpet et gitt tidsrom (vurdere denne opp mot tilsvarende målinger for andre kjøreplanalgoritmer kjørt på de samme datasettene). Ventetiden for en jobb er definert som:

```
ventetid = total tid i systemet – kjøretid
```

dvs. den tiden jobben har ventet for å få tildelt CPU. Med foreslåtte variable for en jobb får vi:

```
ventetid = (ferdigtid - ankomsttid) - kjøretid
```

Data for jobbene skal leses fra to oppgitte tekstfiler TOD063_4_2.data1 og TOD063_4_2.data2. Første tall på filen gir antall jobber. Deretter ligger data om hver jobb linjevis i rekkefølge jobb-nummer, kjøretid, ankomsttid med # som skilletegn. Eksempel på verdier:

Jobbnr.	Ankomsttid [ms]	Kjøretid [ms]
1	0	30
2	2	20
3	4	5
4	8	20
5	10	15

Oppgave 2a (Alle skal utføre denne)

Lag 2 nesten like klasser *KjørePlan1* og *KjørePlan2*, som begge bare inneholder et main()-program (det ene med ordinær *kø*, det andre med *ordnet liste*) som bruker algoritmen beskrevet over og som beregner ventetid for de ulike jobbene og den gjennomsnittlige ventetiden. Begge programmer skal kjøres for data på *begge* de oppgitte filene. Opplysningene skrives ut på skjerm med passende format.

I denne oppgaven må dere starte med å lage de enkelte klassene som behøves, og så lage små test-programmer for å finne ut om hver enkelt klasse er ok før dere går videre til hovedprogrammet. Grense-snittene for klassene skal være enkle, veldefinerte og notasjonsuavhengige (representasjonene skal være skjult) for brukere/klientprogram.
Det skal lages to main() for hver av de to kjøreplanene. Du lager to mapper, en for hver kjøreplan.

Øvrige klasser i denne oppgaven kan være:

Jobb-klasse som også må implementere *interface Comparable* (for å kunne sammenligne to jobber v.h.a. kjøretidene med en metode compareTo(..). Data pr. jobb er jobbnummer, ankomsttid, (krav til) kjøretid og ferdigtid.

Kø og **OrdnaListe** (en av de gitte implementeringene som ligger på It'slearning)

Jobbsamling (med antall og tabell av jobber).

Klassen *Jobbsamling* med data *antall jobber* og tabell av Jobb-objekter kan bl.a. ha følgende metoder:

```
// Legger et nytt Jobb-objekt inn i jobbsamlingen
public void leggTilJobb(Jobb jobb)

// Leser alle rådata om jobber fra en tekstfil og inn i jobbsamlingen
public void lesFraFil(String filnavn)

// Retunerer med første jobb som er ankommet før/ved tidspunkt tid (hvis slike finnes)
public Jobb hentFørstAnkommet(int tid)
```

For kjøreplan1 (FCFS) må vi hente den jobben som har minst ankomsttid <= tid.
For kjøreplan2 (SJN) er det nok å hente ut den første jobben som har ankomsttid <= tid.

```
// Summerer ventetidene til alle jobbene i samlingen
public int finnSumVentetider()
```

Det kan være oversiktlig å benytte to objekter av klasse Jobbsamling i main-programmet, ett for jobb-data som er innlest fra fil, og ett for de samme jobbene etter at de er ferdig utført. For å holde rede på (simulert) klokkeslett, kan du bruke en enkel heltallsvariabal i main() - programmet som teller opp millisekunder.

Krav til dokumentasjon:

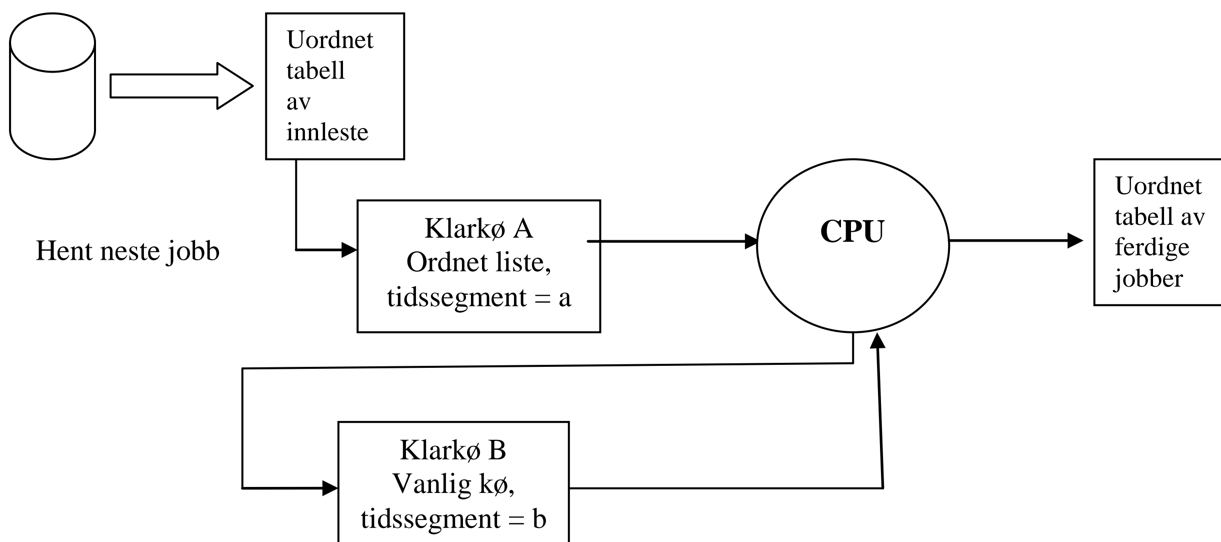
1. Klassediagram (UML-notasjon) som også tar med metoder og egenskaper (data).
2. Figur som viser Java-implementering av Jobb og Jobbsamling (implementering av datastruktur).
3. Kildeprogram med kjøring av data som gitt i oppgaven. For hver kjøring skal følgende data skrives ut: navn på datafil, alle jobbene i den rekkefølge de blir ferdige med data (jobbnummer, ankomsttid, kjøretid, ferdigtid, eventuelt kalkulert ventetid), gjennomsnittlig ventetid.

Eksempel på utskrift fra kjøring:

```
Kjøring med OrdnaListe
Leser inn CPU-jobber fra fil
Jobbsamling opprettet med 10 CPU-jobber

Datafil: TOD063_4_2.data2
Alle tider i ms

Nr      Ankomst      CPU-tid      Ferdig    Ventetid
-------------------------------------------------
 1            0            5           5           0
 4            4           10          15           1
 3            4           25          40          11
 5           40           15          55           0
 6           50           30          85           5
 8           60            8          93          25
 7           60           18         111          33
 10         100           35         146          11
 9           80           40         186          66
 2            0          100         286         186
-------------------------------------------------
Gjennomsnittlig ventetid: 33.8 ms

Gj,sn.ventetid for alle kjøringene: Kjøreplan1 (FCFS) 17.0, 96,2
                                    Kjøreplan2 (SJN)  14,0  33,8
```

Oppgave 2 b (Frivilig ekstraoppgave)

Et alternativ til de to teknikkene du skulle bruke i *KjørePlan1* og *KjørePlan2* er å la alle jobber få litt tid relativt raskt (dermed får korte jobber kort svartid). Hvis en jobb ikke blir ferdig i løpet av denne tiden, stopper vi utføringen og flytter jobben over i en kø med lavere prioritet. Figuren under illustrerer hvordan en bestemt kjøreplanalgoritme virker (forklaring er gitt nedenfor).



En jobb er en sekvens av instruksjoner som skal utføres. Vi forutsetter at en jobb midlertidig kan stoppes etter en instruksjon og senere gjenopptas fra og med neste instruksjon. Dette er realistisk og dere vil senere (i andre fag) lære hvordan dette gjøres i praksis.

Du skal nå lage en klasse *KjørePlan3* med et main-program der det er *to* køer av jobber som er klar for utføring. **Kø A** er en ordnet liste (kap. 8) der jobber med *kortest kjørtetid* står først i listen (ved like kjøretider sammenligner vi ankomsttider). **Kø B** er en vanlig kø (først inn - først ut).

Når CPU-en er ferdig med en jobb (helt eller delvis) sjekker den om nye jobber har ankommet. I så fall blir disse lagt inn i kø A (dvs. den ordnete listen). Deretter velger den ut jobb for utføring etter følgende regler:

1. Hvis det finnes jobber i A, velges den som har høyest prioritet, dvs kortest kjøretid.
2. Hvis det **ikke** finnes jobber i A, velges første jobb i B.

Hvis vi ser på en jobb vil denne gå gjennom følgende faser:

1. ankomst i kø A
2. eventuelt vente i A
3. bli utført inntil a ms
4. hvis jobben ikke er ferdig blir den lagt inn i B(med oppdatert *resttid*)
5. så lenge jobben ikke er ferdig
   - eventuelt vente i B
   - bli utført inntil b ms
   - hvis den ikke er ferdig, blir *resttid* oppdatert og jobben lagt tilbake i B (bakerst)

For å løse problemet på denne måten må du altså legge inn et ekstra datafelt *resttid* i **Jobb**-klassen og ta med nødevendige hent- og sett-metoder. I konstruktøren settes resttid lik kjøretid.

Kjør dette programmet med begge de to gitte datafilene og skriv ut resultatene som tidligere. Test med ulike tidssegmenter a og b (f. eks. verdier mellom 0 og 100). Summer opp resultatene og sammenlign med resultatene fra de to programmene i oppgave 2 a.

Oppgåve 3 (Rekursjon, læreboka kap. 7)

a) Summen av de n første naturlige tall er gitt ved: $S_n = 1+2+3+\ldots+n$

En formel for å finne S_n er gitt ved: $S_n = S_{n-1} + n,\ S_1 = 1$

Lag en rekursiv Java-metode som beregner S_n, og skriv et enkelt hovedprogram som bruker denne metoden for å finne S_{100}.

b) Gitt tallfølgen $\{ a_n \}$ der de enkelte ledd kan finnes med formelen:

$a_n = 5a_{n-1} - 6a_{n-2} + 2$ for $n > 1$ og startkrav $a_0 = 2$, $a_1 = 5$

Lag en rekursiv Java-metode som beregner a_n, og skriv et enkelt hovedprogram som bruker denne metoden til å vise de 10 første leddene i tallfølgen.

c) Franskmannen Edouard Lucas lanserte i 1884 et puslespill som han kalte *tårnene i Hanoi.* Problemet gir en tallfølge $\{ a_n \}$ gitt ved:

$a_n = 2a_{n-1} + 1,\ n > 1$ og $a_1 = 1$

der a_n er antall flyttinger med n antall ringer. Vi kan vise, f.eks. ved induksjon, at en formel som gir verdien av det enkelte ledd er gitt ved:

$a_n = 2^n - 1$

Dersom vi bruker 64 ringer og antar at en manuell flytting krever 1 sekund, så vil det ta ca $5.85 * 10^{11}$ år å flytte alle ringene fra første tårn og over til siste.

I læreboken for faget *Datastrukturer og algoritmer* i kap. 10.3 er det gitt en programkode for dette spillet. Studer dette programmet i boken. Kjør først programmet slik det står i boken (eller på web-siden for faget) med n-verdier 2, 3 og 4. Modifiser deretter koden slik at du *teller opp antal flytt* i stedet for å ta med utskriftssetningen. Kjør denne varianten av programmet for noen ulike verdier av n (antall ringer), for eksempel n mellom 28 og 32, og mål tiden vha. metoder i Date-klassen.

For å måle brukt tid (i millisekunder) med metoder i klassen **Date** tar du med setningen `import.java.util.*;` som første linje i programmet. Du oppretter et objekt som vanlig med (eksempel):

```
Date tidspunkt = new Date()
```

Fra et Date-objekt kan du hente tiden i millisekunder (fra et eller annet tidspunkt for ”veldig lenge siden”) med metoden getTime():

```
long tid_i_millisekunder = tidspunkt.getTime();
```

Ved å registrere to slike tidspunkt, rett før og rett etter kall av den rekursive metoden, kan du finne brukt tid ved å ta differansen mellom tidene (divider på 1000 så får du tiden i sekunder). Skriv ut brukt tid og antall ringer n.

Kontroller at tidsforbruket for ulike n-verdier stemmer overens med løsningen av rekursjonsligningen.

Krav til innlevering for oppgave 3:

- i) Programlisting og utskrift av kjøring av a), b) og c)
- ii) Dokumentasjon av minst 3 ulike tidsforbruk på pkt. c).
- iii) Hvordan stemmer resultatet av antall flyttinger programmet måler med formelen $a_n = 2^n - 1$?