Høgskolen i Telemark

EKSAMEN

5610 ALGORITMAR OG DATASTRUKTURAR

14.05.2012

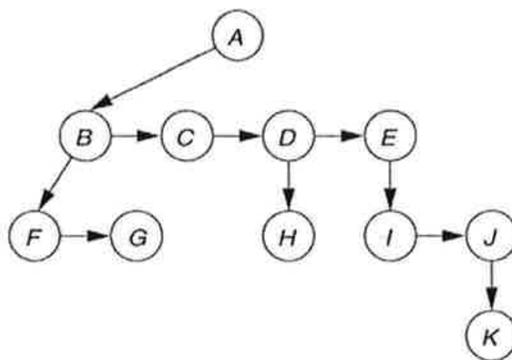
Tid:	9-14 (5 timar)
Målform:	Bokmål og nynorsk
Sidetal:	11 (forside + 5 + 5)
Hjelpemiddel:	Alle trykte og skrivne
Vedlegg:	Ingen

Eksamensresultata blir offentliggjort på nettet via Studentweb

Råd og retningslinjer. Les oppgaveteksten godt før du går i gang med å løse oppgava. Deloppgavene er uavhengige av hverandre i den forstand at om du ikke får til en oppgave, kan du likevel gjøre neste, som om den første var løst. Fordél tida godt på alle oppgavene. Om du mener en oppgave er upresis, så skriv din egen presisering.

Redigerbar trestruktur

Hele oppgavesettet dreier seg om å utvikle klasser for generell redigerbar trestruktur, i betydningen at applikasjonen kan forme treet fritt. Det er altså ikke et søketre. Det er heller ikke et binærtre, hver node kan ha vilkårlig mange barn. Dette skal implementeres ved at hver node har peker til sitt første barn, og dessuten til sin neste søsken. Dette er illustrert i figur 1 nedenfor, hentet fra figur 18.3 i læreboka. Barnpekere og søskenpekere som ikke er vist i figuren, eksempelvis E sin søskenpeker, er null.



Figur 1

I tillegg skal nodene ha en "foreldrepeker" som peker til foreldrenoden, altså oppover i treet. Eksempelvis vil foreldrepekeren til både B, C, D og E peke på A. Endelig skal noden selvsagt ha peker til et dataelement. Rotnoden har ikke forelder og ikke søsken.

Oppgave 1 - Nodeklassen (38%)

Nodeklassen *RNode* skal samsvare med følgende skisse:

```
public class RNode<Type>
{
  // Instansvariable
  ...

  // Konstruktørmetoder
  public RNode(Type element, RNode<Type> forelder){...}
  public RNode(Type element, RNode<Type> forelder,
             RNode<Type> søsken, RNode<Type> barn){...}

  // Andre metoder
  public Type getElement(){...}
  public RNode<Type> getForelder(){...}
  public RNode<Type> getSøsken(){...}
```

```
  public RNode<Type> getBarn(){...}
  public int antBarn(){...}
  public int antSøsken(){...}
  public int etterkommere(){...}
  public int høyde(){...}
  public boolean erRot(){...}
  public boolean erBlad(){...}
  public boolean erEnebarn(){...}
}
```

Oppgave 1a (10%)

Deklarer nødvendige instansvariable i klassen, lag de to konstruktørmetodene, og de fire get-metodene. Metoden *getElement* skal returnere nodens element, *getForelder* skal returnere foreldrenoden, getSøsken skal returnere neste søsken, og *getBarn* skal returnere første barn. Du kan anta at nodeklasse og treklasse ligger i samme pakke og skal ha gjensidig tilgang – bruk pakkebeskyttelse på instansvariable.

Oppgave 1b (10%)

Metoden *antBarn* skal returnere hvor mange barn denne noden har. Metoden *antSøsken* skal returnere hvor mange søsken denne noden har, inklusive seg selv. Lag de to metodene.

Oppgave 1c (11%)

Metoden *etterkommere* skal returnere antall etterkommere av denne noden, inklusive seg selv. Altså antall noder i det subtreet der denne noden er rotnode. Metoden *høyde* skal returnere høyden til noden. Lag de to metodene, bruk rekursjon til begge metodene.

Oppgave 1d (7%)

Metoden *erRot* skal returnere true dersom noden er rotnode, false ellers. Metoden *erBlad* skal returnere true dersom noden ikke har barn, false ellers. Metoden *erEnebarn* skal returnere true dersom noden er eneste barn i søskenflokken, false ellers. Lag de tre metodene.

Oppgave 2 – Treklassen (10%)

Treklassen skal lages i samsvar med følgende skisse.

```
public class RTre<Type> {
  // Instansvariabel
  ...

  // Konstruktørmetode
  public RTre(Type e) {...}

  // Andre metoder
  public int etterkommere(){...}
```

```
  public int høyde() {...}
}
```

Oppgave 2a (10%)

Parameteren til konstruktørmetoden skal bli element i rotnoden. Metoden *etterkommere* returnerer antall noder i hele treet. Metoden *høyde* returnerer høyden på treet. Lag ferdig class *RTre*.

Oppgave 3 – Posisjon / iterasjon (34%)

For å kunne bygge treet slik vi ønsker, må vi kunne «bevege oss» rundt i treet og si at her vil vi sette inn nytt element. Til dette bruk trenger vi en posisjonspeker som kan flyttes. Dette vil ligne mye på iteratorbegrepet, men vi velger her å kalle det en *posisjon*.
Følgende grensesnitt er alt definert:

```
public interface Posisjon<Type> {
  boolean gyldig();
  Type element();
  Posisjon<Type> forelder();
  Posisjon<Type> nesteSøsken();
  Posisjon<Type> forrigeSøsken();
  Posisjon<Type> neste();
  Posisjon<Type> forrige();
  Posisjon<Type> barn();
  Posisjon<Type> rot();
  Posisjon<Type> settInnSøsken(Type element);
  Posisjon<Type> settInnBarn(Type element);
  Type bytt(Type element);
}
```

Det skal lages en klasse *EnkelPosisjon* som implementerer dette grensesnittet. Den skal bare ha én instansvariabel - en peker til en node i et tre. Denne pekeren kan også bli null, da er den ikke *gyldig*. Også *EnkelPosisjon* ligger i samme pakke som *Posisjon*, *RNode* og *RTre*.

Oppgave 3a (8%)

Deklarer instansvariablen, lag de to konstruktørmetodene, og metodene *gyldig* og *element*. Den ene konstruktørmetoden skal få inn en node og ta vare på den, den andre skal få inn et tre og sette posisjonen til treets rotnode. Metoden *gyldig* skal returnere true dersom nodepekeren ikke er null. Metoden *element* skal returnere nodens element. Ved feilsituasjoner – kast *IllegalStateException*.

Oppgave 3b (16%)

Metodene *forelder*, *nesteSøsken*, *forrigeSøsken*, *neste*, *forrige*, *barn* og *rot* har det til felles at de flytter nodepekeren og deretter returnerer seg selv, posisjonsobjektet. Hensikten med dette

er at applikasjonen med kompakt kode kan gjøre flere flytteoperasjoner. Å flytte en posisjon fra en node til nodens tredje barn (eksempelvis fra A til D i figur 1) kan da gjøres med

```
posisjon.barn().nesteSøsken().nesteSøsken();
```

Dette er vanlig stil i enkelte programmeringsmiljøer, men ikke så mye brukt i Java. Metoden *forelder* skal flytte til foreldren (oppover i figur 1), metoden *nesteSøsken* skal flytte til neste søsken (mot høyre i figur 1), metoden *forrigeSøsken* skal flytte til forrige søsken (mot venstre i figur 1). Her i klassen *EnkelPosisjon* skal metoden *neste* gjøre det samme som *nesteSøsken*, og metoden *forrige* skal gjøre det samme som *forrigeSøsken*. Metoden *barn* skal flytte nedover (jmf figur 1) til det første barnet. Metoden *rot* skal flytte til rotnoden. Ved feilsituasjoner – kast *IllegalStateException*.

Lag metodene *forelder*, *nesteSøsken*, *forrigeSøsken*, *neste*, *forrige*, *barn* og *rot*.

Oppgave 3c (10%)

Metoden *settInnSøsken* skal opprette en ny node like til høyre for posisjonspekeren. Resten av søskenflokken flyttes ett hakk mot høyre. Metoden *settInnBarn* oppretter en ny node som første barn av posisjonspekeren. Eventuelle eksisterende barn flyttes ett hakk mot høyre. Begge disse metodene skal flytte posisjonspekeren til den nye noden, og returnere seg selv. Metoden *bytt* skal bytte ut elementet i posisjonspekerens node med det nye, det gamle elementet skal returneres. Ved feilsituasjoner – kast *IllegalStateException*. Lag disse tre metodene.

Oppgave 4 – Preorden iterator (18%)

Som tidligere nevnt, posisjon er en form for iterator. Her skal vi se på klassen PreordenPosisjon, se følgende skisse:

```
class PreordenPosisjon<Type> extends EnkelPosisjon<Type> {
  // Konstruktørmetoder
  ...

  // Andre metoder
  public Posisjon<Type> neste(){...}
  public Posisjon<Type> forrige(){...}
}
```

Metodene *neste* og *forrige* planlegges her reimplementert slik at de flytter til neste og forrige i preorden rekkefølge.

Oppgave 4a (8%)

Lag metoden *neste*. Du skal ikke lage metoden *forrige*.

Hint: Dersom noden har barn, gå til første barn. Ellers: Dersom noden har søsken, gå til neste søsken. Ellers: Se om foreldrenoden har søsken, eller om besteforeldrenoden har søsken, osv.

Oppgave 4b (10%)

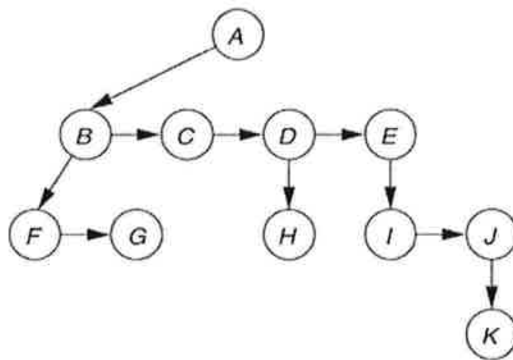
Skriv en main-metode som oppretter et RTre av String'er slik at det har samme struktur og innhold som treet i figur 1. main-metoden ligger ikke i samme pakke som de klassene vi har sett på.
Deretter skal main-metoden traversere treet i preorden rekkefølge og skrive ut innholdet av nodene.
Helt til slutt skal du oppgi hvordan den utskriften vil se ut – oppgi nodene fra figur 1 i preorden rekkefølge.

Lykke Til!

Råd og retningsliner. Les oppgåveteksten godt før du går i gang med å løyse oppgåva. Deloppgåvene er uavhengige av kvarandre i den meining at om du ikkje får til ei oppgåve, kan du likevel gjere neste, som om den fyrste var løyst. Fordél tida godt på alle oppgåvene. Om du meiner ei oppgåve er upresis, så skriv din eigen presisering.

Redigerbar trestruktur

Heile oppgåvesettet dreier seg om å utvikle klasser for generell redigerbar trestruktur, i tydinga at applikasjonen kan forme treet fritt. Det er altså ikkje eit søketre. Det er heller ikkje eit binærtre, kvar node kan ha vilkårleg mange barn. Dette skal implementerast ved at kvar node har peikar til sitt fyrste barn, og dessutan til sin neste sysken. Dette er illustrert i figur 1 nedanfor, henta frå figur 18.3 i læreboka. Barnpeikarar og syskenpeikarar som ikkje er vist i figuren, til dømes E sin syskenpeiker, er null.



Figur 1

I tillegg skal nodane ha ein "foreldrepeikar" som peikar til foreldrenoden, altså oppover i treet. Til dømes vil foreldrepeikaren til både B, C, D og E peike på A. Endeleg skal noden sjølsagt ha peikar til eit dataelement. Rotnoden har ikkje forelder og ikkje sysken.

Oppgåve 1 – Nodeklassen (38%)

Nodeklassen *RNode* skal samsvare med fylgjande skisse:

```
public class RNode<Type>
{
  // Instansvariable
  ...

  // Konstruktørmetodar
  public RNode(Type element, RNode<Type> forelder){...}
  public RNode(Type element, RNode<Type> forelder,
               RNode<Type> sysken, RNode<Type> barn){...}

  // Andre metodar
  public Type getElement(){...}
  public RNode<Type> getForelder(){...}
```

```
  public RNode<Type> getSysken(){...}
  public RNode<Type> getBarn(){...}
  public int antBarn(){...}
  public int antSysken(){...}
  public int etterkomarar(){...}
  public int høgde(){...}
  public boolean erRot(){...}
  public boolean erBlad(){...}
  public boolean erEinebarn(){...}
}
```

Oppgåve 1a (10%)

Deklarer naudsynte instansvariable i klassa, lag dei to konstruktørmetodane, og dei fire get-metodane. Metoden *getElement* skal returnere noden sitt element, *getForelder* skal returnere foreldrenoden, getSysken skal returnere neste sysken, og *getBarn* skal returnere fyrste barn. Du kan gå ut i frå at nodeklasse og treklasse ligg i same pakke og skal ha gjensidig tilgang – bruk pakkebeskyttelse på instansvariable.

Oppgåve 1b (10%)

Metoden *antBarn* skal returnere kor mange barn denne noden har. Metoden *antSysken* skal returnere kor mange sysken denne noden har, inklusive seg sjøl. Lag dei to metodane.

Oppgåve 1c (11%)

Metoden *etterkomarar* skal returnere antal etterkomarar av denne noden, inklusive seg sjøl. Altså antal nodar i det subtreet der denne noden er rotnode. Metoden *høgde* skal returnere høgda til noden. Lag dei to metodane, bruk rekursjon til begge metodane.

Oppgåve 1d (7%)

Metoden *erRot* skal returnere true dersom noden er rotnode, false ellers. Metoden *erBlad* skal returnere true dersom noden ikkje har barn, false ellers. Metoden *erEinebarn* skal returnere true dersom noden er eineste barn i syskenflokken, false ellers. Lag dei tre metodane.

Oppgåve 2 – Treklassa (10%)

Treklassa skal lagast i samsvar med fylgjande skisse.

```
public class RTre<Type> {
  // Instansvariabel
  ...

  // Konstruktørmetode
  public RTre(Type e) {...}

  // Andre metodar
```

```
  public int etterkomarar(){...}
  public int høgde() {...}
}
```

Oppgåve 2a (10%)

Parameteren til konstruktørmetoden skal bli element i rotnoden. Metoden *etterkomarar* returnerer antal nodar i heile treet. Metoden *høgde* returnerer høgda på treet. Lag ferdig class *Rtre*.

Oppgåve 3 – Posisjon / iterasjon (34%)

For å kunne bygge treet slik vi ynskjer, må vi kunne «flytte oss» rundt i treet og seie at her vil vi sette inn nytt element. Til dette bruk treng vi ein posisjonspeikar som kan flyttast. Dette vil likne mykje på iteratoromgrepet, men vi vel her å kalle det ein *posisjon*.
Fylgjande grensesnitt er alt definert:

```
public interface Posisjon<Type> {
  boolean gyldig();
  Type element();
  Posisjon<Type> forelder();
  Posisjon<Type> nesteSysken();
  Posisjon<Type> forrigeSysken();
  Posisjon<Type> neste();
  Posisjon<Type> forrige();
  Posisjon<Type> barn();
  Posisjon<Type> rot();
  Posisjon<Type> settInnSysken(Type element);
  Posisjon<Type> settInnBarn(Type element);
  Type bytt(Type element);
}
```

Det skal lagast ei klasse *EnkelPosisjon* som implementerer dette grensesnittet. Den skal berre ha ein instansvariabel - ein peikar til ein node i eit tre. Denne peikaren kan også bli null, då er den ikkje *gyldig*. Også *EnkelPosisjon* ligg i same pakke som *Posisjon*, *RNode* og *RTre*.

Oppgåve 3a (8%)

Deklarer instansvariablen, lag dei to konstruktørmetodane, og metodane *gyldig* og *element*. Den eine konstruktørmetoden skal få inn ein node og ta vare på den, den andre skal få inn eit tre og sette posisjonen til treets rotnode. Metoden *gyldig* skal returnere true dersom nodepeikaren ikkje er null. Metoden *element* skal returnere noden sitt element. Ved feilsituasjonar – kast *IllegalStateException*.

Oppgåve 3b (16%)

Metodane *forelder*, *nesteSysken*, *forrigeSysken*, *neste*, *forrige*, *barn* og *rot* har det til felles at dei flyttar nodepeikaren og deretter returnerer seg sjøl, posisjonsobjektet. Meininga med dette

er at applikasjonen med kompakt kode kan gjere fleire flytteoperasjonar. Å flytte ein posisjon frå ein node til nodens tredje barn (til dømes frå A til D i figur 1) kan då gjerast med `posisjon.barn().nesteSysken().nesteSysken();`
Dette er vanleg stil i nokre programmeringsmiljø, men ikkje så mykje bruka i Java. Metoden *forelder* skal flytte til foreldren (oppover i figur 1), metoden *nesteSysken* skal flytte til neste sysken (mot høgre i figur 1), metoden *forrigeSysken* skal flytte til forrige sysken (mot venstre i figur 1). Her i klassen *EnkelPosisjon* skal metoden *neste* gjere det same som *nesteSysken*, og metoden *forrige* skal gjere det same som *forrigeSysken*. Metoden *barn* skal flytte nedover (jmf figur 1) til det fyrste barnet. Metoden *rot* skal flytte til rotnoden. Ved feilsituasjonar – kast *IllegalStateException*.

Lag metodane *forelder*, *nesteSysken*, *forrigeSysken*, *neste*, *forrige*, *barn* og *rot*.

Oppgåve 3c (10%)

Metoden *settInnSysken* skal opprette ein ny node like til høgre for posisjonspeikaren. Resten av syskenflokken flyttast eitt hakk mot høgre. Metoden *settInnBarn* opprettar ein ny node som fyrste barn av posisjonspeikaren. Eventuelle eksisterande barn flyttast eitt hakk mot høgre. Begge desse metodane skal flytte posisjonspeikaren til den nye noden, og returnere seg sjøl. Metoden *bytt* skal bytte ut elementet i posisjonspeikaren sin node med det nye, det gamle elementet skal returnerast. Ved feilsituasjonar – kast *IllegalStateException*. Lag desse tre metodane.

Oppgåve 4 – Preorden iterator (18%)

Som tidlegare nemnt, posisjon er ein form for iterator. Her skal vi sjå på klassa PreordenPosisjon, sjå fylgjande skisse:

```
class PreordenPosisjon<Type> extends EnkelPosisjon<Type> {
  // Konstruktørmetodar
  ...

  // Andre metodar
  public Posisjon<Type> neste(){...}
  public Posisjon<Type> forrige(){...}
}
```

Metodane *neste* og *forrige* planleggast her reimplementert slik at dei flytter til neste og forrige i preorden rekkefylgje.

Oppgåve 4a (8%)

Lag metoden *neste*. Du skal ikkje lage metoden *forrige*.

Hint: Dersom noden har barn, gå til fyrste barn. Elles: Dersom noden har sysken, gå til neste sysken. Elles: Sjå om foreldrenoden har sysken, eller om besteforeldrenoden har sysken, osv.

Oppgåve 4b (10%)

Skriv ein main-metode som oppretter eit RTre av String'ar slik at det har same struktur og innhald som treet i figur 1. main-metoden ligg ikkje i same pakke som dei klassene vi har sett på.
Deretter skal main-metoden traversere treet i preorden rekkefylgje og skrive ut innhaldet av nodane.
Heilt til slutt skal du oppgje korleis den utskrifta vil sjå ut – oppgje nodane frå figur 1 i preorden rekkefylgje.

Lykke Til!