Lenkelister og beholdere av lenkelister

Et notat for INF1010

Stein Michael Storleer

17. mars 2014

2014-utgaven av notatet er utvidet med programeksemplene som brukes i forelesningene om lenkelister og om rekursjon mellom objekter.

Lister er den vanligste datastrukturen. Vi treffer på den overalt. Når vi har mange objekter i et program, trenger vi en grei måte å holde styr på dem. Å legge dem i ei liste er det vanligste. Og ofte det enkleste også.

Fra den virkelige verden kjenner vi til lister i mange sammenhenger. Tenk bare på spilleliste, deltakerliste, handleliste, savnetliste, sjekkliste, ... Det alle lister har felles, er at det er en sekvensiell ordning av objekter (ord, navn, personer ...), altså en ordning der ett objekt kommer først, så kommer et neste osv. til vi kommer til det siste objektet. Er lista skriftlig, vil vi skrive ordene som står for objektene (f.eks. ord for varene vi skal handle: salt, ost, egg, brød) etter hverandre, eller under hverandre. Når vi tegner datastrukturen liste i et javaprogram gjør vi det ofte på en lignende måte. Lister har vanligvis minst to objekter. Hvis lista kun inneholder ett objekt vil vi nok ikke kalle det ei liste, men i et javaprogram kan ei liste inneholde fra ingen til mange objekter.

I programmering ser vi også lister «overalt». Et javaprogram er ei liste (en sekvens) av setninger. En programlinje (eller en hvilken som helst tekstlinje) er ei liste av tegn. Hele programmet er ei liste av linjer, m.a.o. ei liste av lister av tegn. En fil er ei liste av bytes. En byte er ei liste av bit. Datamaskinas hukommelse er ei liste (en array) av «ord». Sånn kunne vi fortsette.

Arrayer er den datastrukturen vi tenker på først når vi skal lage ei liste i et javaprogram. Det er ofte det greieste også. I tillegg til den sekvensielle ordningen, får vi også en indeks, slik at vi kan lett kan finne et objekt på en bestemt plass i lista.

```
class Ord {
  String ordet;

  Ord(String o) {
    ordet = o;
}

class Ordliste {
  Ord [] ordliste;

  Ordliste(String [] ordene) {
    ordliste = new Ord[ordene.length];
    for (int i = 0; i < ordene.length; i++)
         ordliste[i] = new Ord(ordene[i]);
  }
}
```

```
class NoenOrd {
  public static void main(String[] kommandolinjeord) {
     new Ordliste(kommandolinjeord);
   }
}
```

Programmet ovenfor legger ordene (tekststrengene) vi skriver etter java NoenOrd i tekstarrayen kommandolinjeord når vi skal kjøre programmet.

Oppgave 1 Test programmet. Hva skjer når vi ikke skriver noenting etter java NoenOrd? Test programmet ved å legge til en utskrift av ordene i ordliste i omvendt rekkefølge (snu lista).

Hvis antall objekter i lista varierer mye under programutførelsen, er det to ulemper med en array.

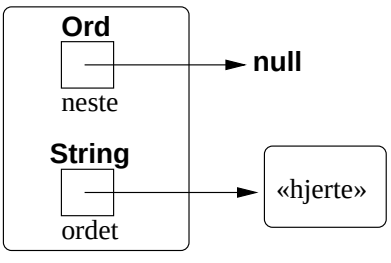
Arrayen må ha en bestemt lengde. Det er ikke alltid vi vet på forhånd hvor stor plass vi trenger og ofte må vi sette av langt mer plass enn det som behøves fordi vi vil være sikre. Likevel kan det vise seg at det ikke er nok og programmet kræsjer.

Hvis vi tar bort objekter i en array, oppstår det et «hull» (i form av en nullpeker). Får vi mange hull blir arrayen «glissen» og vi bør vurdere å flytte objektene (til samme eller en ny array) slik at vi får en sammenhengende del av arrayen som ikke peker på null.

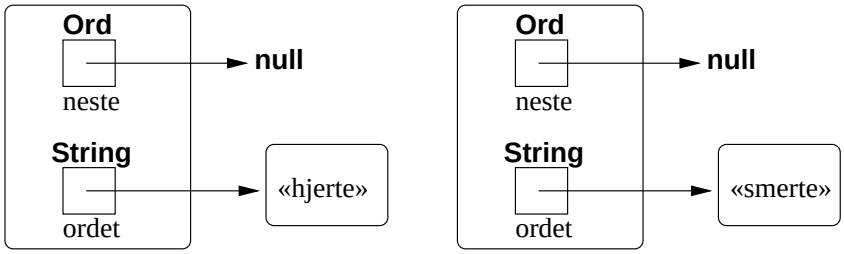
Begge disse problemene unngår vi hvis vi istedet velger å lagre objektene i ei lenkeliste.

Et tredje problem oppstår hvis vi ønsker at objektene skal ha en ordning (være sortert). Da er array uegnet hvis vi ønsker å opprettholde ordningen etterhvert som vi setter inn objekter. Da må vi hele tida flytte på objekter når et nytt objekt skal inn mellom to objekter som ligger «ved siden av hverandre» i arrayen. Den greieste måten å gjøre dette på med en array, er å sette inn alle objektene uordnet, og så sortere dem når alle er kommet på plass. Det fungerer dessverre bare hvis man ikke senere skal sette inn flere objekter. Også dette problemet forsvinner med ei lenkeliste. Da er det «uendelig plass» mellom to objekter som ligger side om side.

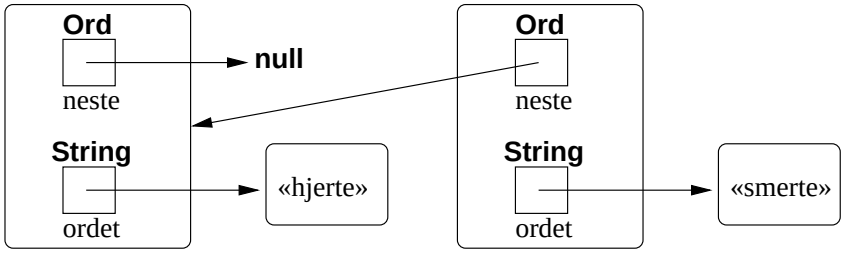
Lenkelister (linked lists, lenket liste, pekerkjedelister, pekerlister, kjedelister) lages ved at objektene som skal legges i lista utstyres med en peker (lenke) som kan peke på påfølgende (neste) objekt. I sin enkleste form, en peker av samme type som objektene. (Senere skal vi se hvordan vi kan gjøre dette når objektene ikke er instanser av samme klasse). Vi tegner dette slik:



Her har vi et objekt (instans) av klassen ord (hvordan kunne vi tydeliggjort det i tegningen?). Det inneholder tekststrengen «hjerte». Merk at ord slik den defineres nedenfor også har en konstruktør som ikke er tegnet inn. String-objektet som ordet peker på er også sterkt forenklet. I dette notatet vil en variabel alltid være tegnet med typen *over* firkanten og navnet *under*. Hvis vi nå lager et objekt til (med tekststrengen «smerte») får vi denne tilstanden i datastrukturen:



Hvis vi nå ønsker å lage ei lenkeliste av de to ordene («smerte» før «hjerte») får vi det til ved å la nestepekeren i «smerte» peke på ordobjektet til «hjerte» slik:

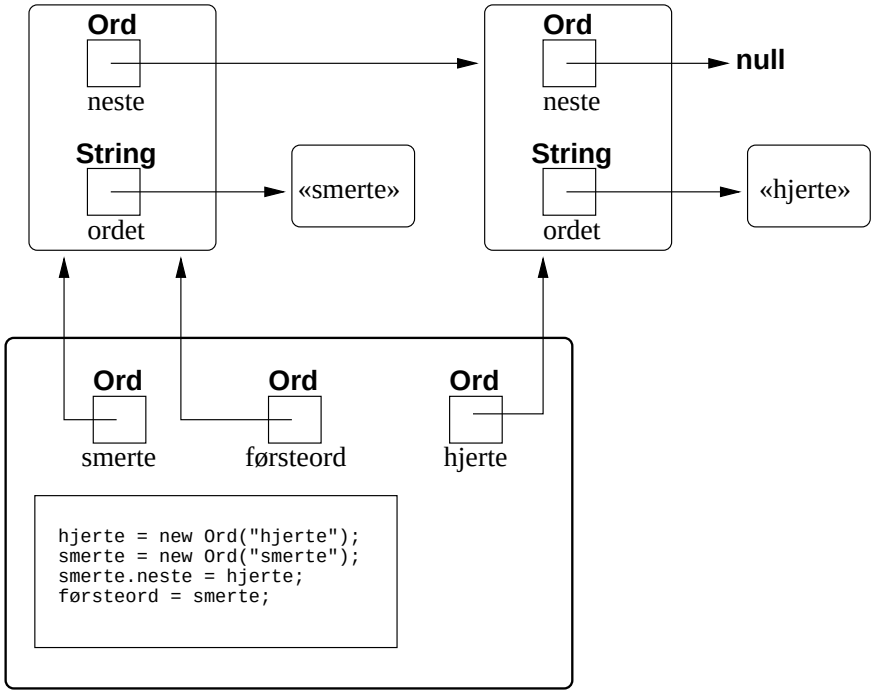


Her er et mulig program som gjør dette.

```
class Ord {
    String ordet;
    Ord neste; // peker til neste ord

    Ord(String o) {
        ordet = o;
        neste = null;
    }
}

class Ordliste {
    Ord smerte, hjerte;
    Ord førsteord;

    Ordliste() {
        hjerte = new Ord("hjerte");
        smerte = new Ord("smerte");
        smerte.neste = hjerte;
        førsteord = smerte;
    }
}
```

```
class Hjerte {
    public static void main (String[] a) {
        new Ordliste();
        // tilstandspåstand se figur nedenfor
    }
}
```

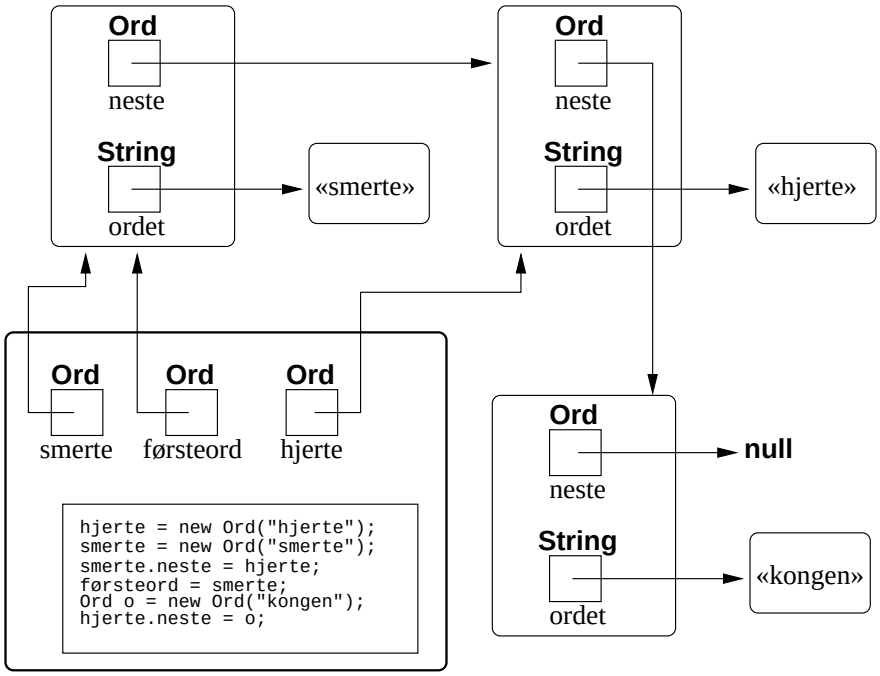
Følgende tegning av ordlisteobjektet illustrerer programmererens påstand om tilstanden i datastrukturen etter at main-metoden har sagt new Ordliste();



Nå kan vi f.eks. legge nye objekter inn i lista ved å utvide konstruktøren (firkantramma i figuren) med:

```
Ord o = new Ord("kongen");
hjerte.neste = o;
```

Pekertilordningen hjerte.neste = o er det lurt å «si» slik når vi programmerer og tegner: «hjerte sin neste settes til **å peke på det samme som** o peker på». Tilstandspåstand etter at konstruktøren er utført:



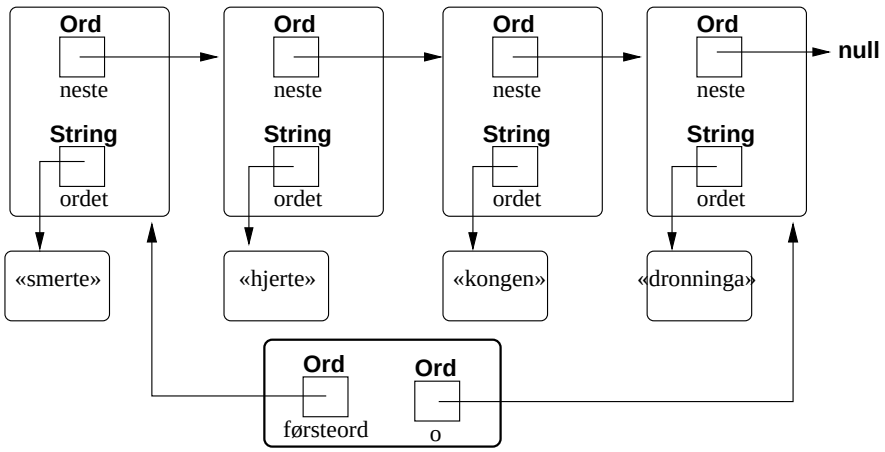
Hvorfor har vi ikke tegnet variabelen Ord o? Slik kan vi fortsette å sette inn så mange objekter vi vil.

```
Ord o = new Ord("dronninga");
hjerte.neste = o;
```

Men var det denne tilstanden vi ville oppnå? Når vi tegner datastrukturen (prøv!) ser vi at det er brudd i lista. Vi mistet «kongen». Med mange objekter er det uhensiktsemssig å ha en variabel som peker på hvert enkelt element. Det holder faktisk med en. Den kaller vi ofte førsteelement eller ganske enkelt første (first). (Noen ganger kalles den liste fordi den holder i (peker på) hele lista. Det vil vi unngå her, fordi vi foretrekker å ha et eget objekt (her Ordliste som er listeobjektet).) La oss da skrive om programmet og legge inn de fire ordene i rekkefølgen «smerte», «hjerte», «kongen», «dronninga»:

```
Ord o = new Ord("smerte");
førsteord = o;
o = new Ord("hjerte");
førsteord.neste = o;
o = new Ord("kongen");
førsteord.neste.neste = o;
o = new Ord("dronninga");
førsteord.neste.neste.neste = o;
```

Vi tegner datastruktur (tilstandspåstand):



Men slik kan vi ikke gjøre det hvis vi har tusenvis av ord vi skal sette inn. Hva om vi setter inn det nye objektet først? Da er det enklere å sette inn objektene i omvendt rekkefølge siden det objektet vi setter inn først da havner bakerst og det objektet vi setter inn sist blir liggende først:

```
Ord o = new Ord("dronninga");
førsteord = o;
o = new Ord("kongen");
o.neste = førsteord;
førsteord = o;
o = new Ord("hjerte");
o.neste = førsteord;
førsteord = o;
o = new Ord("smerte");
o.neste = førsteord;
førsteord = o;
```

Tilstanden i datastrukturen påstår vi nå er den samme som i tegninga ovenfor. Men er vi sikre på det? Hvordan kan vi kontrollere det? Vi lager en metode i listeobjektet (i klassen Ordliste) som skriver ut alle ordene i den rekkefølgen de ligger i lenkelista:

```
public void skrivUtAlleOrdeneIlista() {
  for (Ord o = førsteord; o != null; o = o.neste)
      System.out.println(o.ordet);
}
```

Ordlista vår lages i konstruktøren i Ordliste. Det er lite fleksibelt. La oss i stedet skrive om klassen slik at den tilbyr en metode til å sette inn objekter og som kan kalles utenfra:

```
class Ordliste {
    private Ord førsteord = null;

    public void settInnOrd(Ord o) {
       if (førsteord == null) førsteord = o;
       else
          { o.neste = førsteord;
            førsteord = o; }
    }
```

```
    public void skrivUtAlleOrdeneIlista() {
        for (Ord o = førsteord; o != null; o = o.neste)
            System.out.println(o.ordet);
    }
}
```

Vi kan nå sette inn ord i lista slik

```
Ordliste ordliste = new Ordliste();
ordliste.settInnOrd(new Ord("knekten"));
ordliste.settInnOrd(new Ord("dama"));
ordliste.settInnOrd(new Ord("kongen"));
```

Siden pekeren førsteord er utilgjengelig utenfra, er det bare ved kall på settInnOrd vi nå kan manipulere (her bare sette inn) lista. Ved å gjøre innsetting mulig på bare én måte har vi også bestemt rekkefølgen på ordene. Den vil bli omvendt av rekkefølgen av settInnOrd-kallene. Det vil med andre ord si, at når vi kaller på skrivUtAlleOrdenelista blir utskriftsrekkefølgen omvendt sammenlignet med rekkefølgen på settinnkallene. For programmet ovenfor blir utskriften fra skrivutmetoden i rekkefølgen: kongen, dama, knekten.

I lista er det også behov for å fjerne objekter. I klassen Ordliste er det enklest å gjøre det først i lista, altså ved å ta ut det objektet som førsteord peker på. Det er det ordet som ble satt inn sist:

```
public Ord taUtOrd() {
    Ord ord = førsteord;
    if (ord != null)
        førsteord = ord.neste;
    return ord;
}
```

Stabler og køer. LIFO og FIFO

Da har vi en metode for å sette inn, og en for å ta ut ord. Når vi har bestemt hvordan har vi bestemt rekkefølgen ord kommer ut av lista i forhold til rekkefølgen de ble satt inn. I vårt eksempel blir ordet som sist ble satt inn tatt ut først. **Sist inn først ut**. På engelsk **last in first out: LIFO**. Vi har defiert en *LIFO-køordning* av lista. Merk at ordet *sist/last* her ikke har noe med endene av lista å gjøre, men viser til *når* ordet ble lagt inn. Legg også merke til at det ikke er lista selv som er LIFO. Det bestemmes av metodene som setter inn og tar ut objekter. LIFO-køordning er så vanlig i databehandling at den har fått et eget navn. Ei liste som administreres slik kalles en **stabel**, engelsk **stack**. De to metodene for å legge inn og ta ut fra en stabel har også fått egne navn på engelsk:

```
class OrdStack {
    Ord førsteord;

    public void push(Ord o) { // setter inn
        o.neste = førsteord;
        førsteord = o;
    }
```

```
    public Ord pop() {   // tar ut
        Ord o = førsteord;
        if (o != null) førsteord = o.neste;
        return o;
    }
}
```

Køer i dagliglivet administreres mer «rettferdig» enn en stabel, ved at elementet som har ligget lengst tas ut først. **Først inn først ut** eller **first in first out: FIFO**. Når vi står i kø for å få kjøpt den siste iPaden setter vi pris på at køen/lista administreres slik. For å få til dette må vi i den datastrukturen vi har valgt i Ordliste lete oss fram til det ordet som har ligget lengst. Dette er både tidkrevende og vanskelig å få til. Vanskeligheten består i at metoden skal virke på lister med null, en, to eller fler elementer. Og for å «hekte av» det siste elementet, trenger vi tilgang til nestepekeren i elementet nest sist i lista!

Oppgave 2 Utvid klassen Ordliste ovenfor med metoden public Ord taUtSisteOrd() som fjerner siste element i ordlista.

Ved å utvide datastrukturen i lista slik at vi har en peker til begge endene blir dette langt greiere. For å unngå problemet med å ta ut sist i lista, er det enklest å snu det hele. Vi har en peker til det første ordet (førsteord), en til det siste (sisteord). Ved å sette inn *etter* elementet sisteord peker på og ta ut elementet førsteord peker på, oppnår vi en FIFO-køordning. Ved å gjøre det slik trenger vi ikke gjøre endringer i taUtOrd:

```
class Ordliste {
    private Ord førsteord = null;
    private Ord sisteord = null;

    public void settInnOrd(Ord o) {
        if (førsteord == null) {
            førsteord = o;
            sisteord = o;
        } else {
            sisteord.neste = o;
            sisteord = o;
        }
    }

    public Ord taUtOrd() {
        Ord ord = førsteord;
        if (ord != null)
            førsteord = ord.neste;
        return ord;
    }

    ....
}
```

Av og til vil vi ta ut elementer som hverken har ligget kortest eller lengst i lista. Slik køadministrasjon har ikke egne navn. Unntaket er *prioritetskø* hvor elementene tas ut etter et sorteringskriterium. I INF1010 bør kunne disse to:

- Tar vi alltid ut elementet som har ligget lengst i lista har vi en FIFO-kø («vanlig kø»).
- Tar vi alltid ut elementet som har ligget kortest i lista har vi en LIFO-kø (stabel/stack).

Det er selvfølgelig ingenting i veien for å tilby både LIFO- og FIFO-uttak i samme lenkeliste:

```
class Ordliste {
    ....

    public Ord taUtOrdLIFO() {
        Ord ord = førsteord;
        if (ord != null)
            førsteord = ord.neste;
        return ord;
    }

    public Ord taUtOrdFIFO() {
        Ord ord = førsteord;
        if (førsteord == null) {
            // Tilstandspåstand 1
        }
        else if (førsteord == sisteord) {
            // Tiltandspåstand 2
            førsteord = null;
            sisteord = null;
            // Tiltandspåstand 3
        }
        else {
            // Tiltandspåstand 4
            while (ord.neste != sisteord)
                ord = ord.neste;
            // Tilstandspåstand 5
            sisteord = ord;
            ord = sisteord.neste;
            sisteord.neste = null;
            // Tilstandspåstand 6
        }
        return ord;
    }

    ....
}
```

Tautmetoden FIFO ble langt mer komplisert enn vi skulle tro. Grunnen til dette, er at vi ikke har noen peker til objektet foran objektet vi skal ta ut. (Derfor er det alltid enklest å lage lenkelista slik at vi tar ut først). Tilstandspåstandene om datastrukturen med tilhørende tegninger er nødvendige for at vi skal få dette til riktig.

Tilstandspåstand 1: Lista er tom og ord == null (som er rett returverdi da)

Tilstandspåstand 2: Lista inneholder ett ord og alle tre pekere ord, førsteord og sisteord peker på dette.

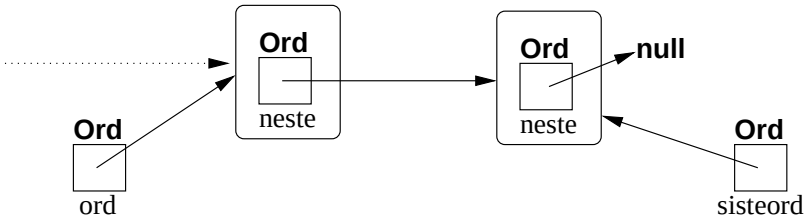
Tilstandspåstand 3: Lista er tom og ord peker på ordet som skal returneres

Tilstandspåstand 4: Lista inneholder mer enn ett ord. ord peker på det første.

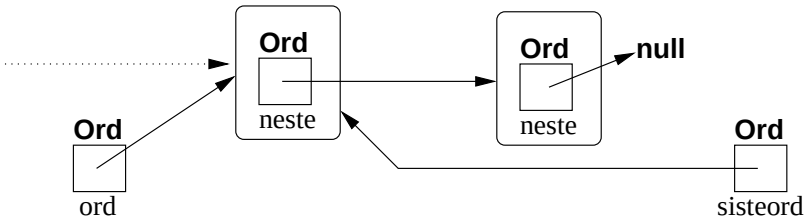
Tilstandspåstand 5: Lista har mer enn ett ord og ord peker på det nest siste

Tilstandspåstand 6: sisteord peker på det som var nest siste ord, og er blitt ett ord kortere og ord peker på det som var det siste (som er rett returverdi)

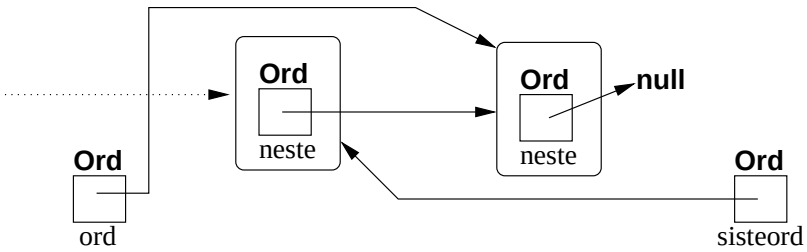
Spranget fra påstand 5 til 6 er det vanskeligste å overbevise oss om. La oss først tegne datastrukturen som berøres av metoden. Ordobjektene har bare med nestepekerne og den stiplede pila er enten pekeren fra førsteordvariabelen, eller nestepekeren i objektet foran. Her er datastrukturen som svarer til tilstandspåstand 5:



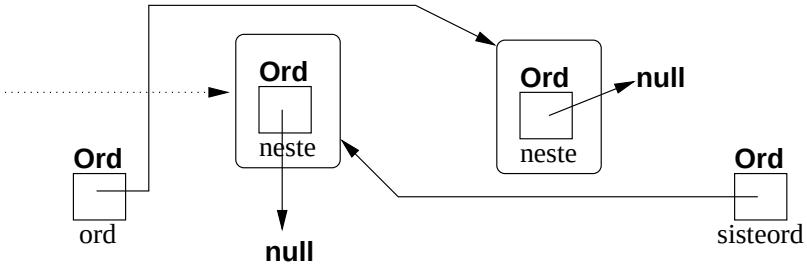
Først utføres tilordningen sisteord = ord; Tilstanden blir da:



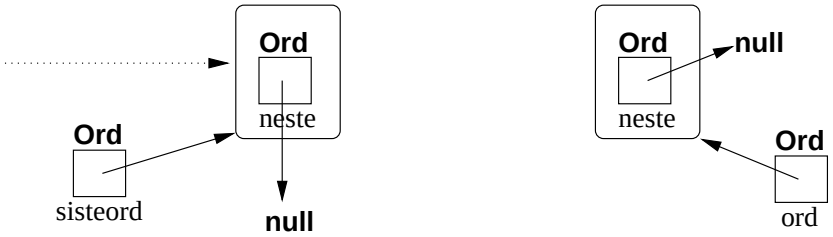
Dernest ord = sisteord.neste;



Tilslutt sisteord.neste = null; Da blir ordet som skal ut «hektet av»:



Vi flytter litt på objektene (men ikke pekerne!) og har situasjonen (samme som over):



Og vi ser at tilstandspåstand 6 holder. Legg også merke til at førsteord ikke berøres av disse tilordningene.

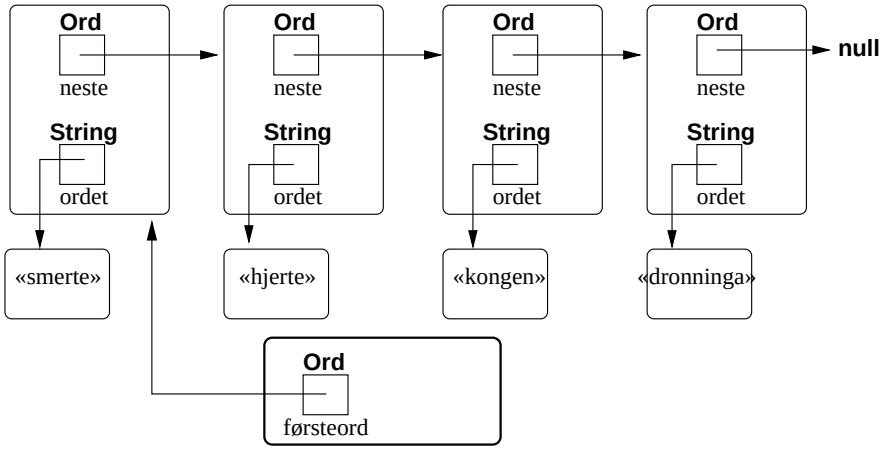
Tilstandspåstander er påstander om tilstanden i (deler av) datastrukturen. Tilstandspåstand 5 kan også skrives mer presist som sisteord.neste == null & ord.neste == sisteord. Av dette følger (logisk) at ord peker på det nest siste ordobjektet fordi det bare er siste ordet i lista som har en nestepeker som er null.

Vi kunne spart oss en del arbeid i tautmetoden, hvis vi hadde hatt en peker til som pekte på nest siste objekt. På den annen side hadde da settinnmetoden blitt mer komplisert fordi vi måtte behandle lister med null, ett og to objekter spesielt.

Oppgave 3 Utvid klassen med en peker som peker på nest siste objekt. Skriv om metodene deretter. Pass særlig på tilfeller der lista inneholder mindre enn tre elementer. Bruk tilstandspåstander.

Oppgave 4 Utvid datastrukturen i listeobjektet (ordliste) med en heltallsvariabel som inneholder antall ord i lista. Blir noen av metodene enklere å programmere?

I en array kan vi ganske enkelt ta ut objekter «midt i» arrayen. Det er ikke bestandig vi får tak i det vi vil først eller sist. Hvordan skal vi få til dette i lenkelista? Først må vi vite hvilket objekt vi vil ta ut. La oss ta utgangspunkt i lista:



Gitt at vi ønsker å fjerne ordobjektet som peker på «kongen». Hvordan finner vi det? Vi lager en metode finnOrdMedString i ordlisteobjektet som returnerer en peker til ordobjektet hvis det finnes. Parameter blir ((String s), og returtype Ord:

```
public Ord finnOrdMedString(String s){
    boolean funnet = false;
    Ord ord = førsteord;
    String o = null;
    while (ord != null & ! funnet) {
        // tilstandspåstand 10
        o = ord.ordet;
        if ( o.equals(s) ) funnet = true;
        else ord = ord.neste;
    }
    // tilstandspåstand 11
    if (! funnet) ord = null;
    return ord;
}
```

Når vi lager en while- eller for-løkke, må vi alltid passe godt på. Vi får veldig fort *nullpekerfeil* fordi vi forsøker å inspisere et objekt som ikke eksisterer! Også her er det meget nyttig å tenke tilstandspåstander:

Tilstandspåstand 10: ord peker ikke på null og funnet er false (pga. betingelsen i while)

Tilstandspåstand 11: ord peker på null eller funnet er true eller begge deler (også pga. betingelsen i while)

Ved tilstandspåstand 11 tester vi om funnet er false. Hvis ja, må while-løkka ha stoppet fordi ord == null og vi kan konkludere med at ordet ikke fantes i lista. Vi setter ord = null; (returverdien). Hvis nei, må løkka ha stoppet fordi funnet == true. Dette må ha skjedd fordi o.equals(s) == true, og siden o == ord.ordet må ord peke på ordobjektet med tekststrengen vi leter etter (s). Før vi går videre er det også lurt og sjekke at metoden virker korrekt også når lista er tom og når den inneholder kun ett objekt.

Testen o.equals(s) trenger kanskje en forklaring. equals er en metode definert i klassen String som er slik at o.equals(s) == true hvis og bare hvis o og s er nøyaktig samme tekststreng. (Se definisjonen av String i Javas API).

Kallet Ord funnetOrd = finnOrdMedString(«kongen»); (« og » betyr det samme som dobbelt anførselstegn") skal hvis metoden ovenfor virker som planlagt, gjøre at funnetOrd peker på ordobjektet (med ordet lik) «kongen». Dette objektet skal ut. Men for å få dette objektet ut av lista, må vi ha en peker til ordobjektet «hjerte», fordi det er «hjerte»s nestepeker som skal skifte verdi. Men strevet med finnOrdMedString er ikke forgjeves! Vi skriver den om, slik at den returnerer en peker til ordet som har en nestepeker som peker på ordobjektet vi leter etter.

Oppgave 5 Skriv metoden finnOrdetForanOrdMedString(String s) slik at den returnerer en peker til ordet som har en nestepeker som peker på ordobjektet vi leter etter. Pass særlig på at den virker riktig når lista er tom eller meget kort. Hva skal metoden returnere hvis ordet det letes etter er i objektet som pekes på av førsteord eller av førsteord.neste? Bruk tilstandspåstander og tegninger.

Vi fortsetter med å programmere metoden taUtOrdMedString uten å finne den først:

```
public Ord taUtOrdMedString(String s) {
    Ord ordfjernet = null;
    Ord ord = førsteord;
    if (ord == null) {} // lista er tom. ordfjernet == null
    else if (førsteord.ordet.equals(s)) {
         // Spesialtilfelle:
         // førsteord peker på ordet som skal fjernes
         førsteord = førsteord.neste; // fjernet!
         ordfjernet = ord; // ord peker på det som var første
    }
    else if (førsteord.neste == null) {}
         // ett ord i lista. Ikke det vi leter etter
    else if (førsteord.neste.ordet.equals(s)) {
         // ord nr to skal fjernes :
         ordfjernet = førsteord.neste;
         førsteord.neste = førsteord.neste.neste;
    }
    else {
         while (ord.neste != null && ! ord.neste.ordet.equals(s))
             ord = ord.neste;
         // Tilstandspåstand 20
         if (ord.neste != null) {
             ordfjernet = ord.neste;
             ord.neste = ord.neste.neste;
         }
    }
    return ordfjernet;
}
```

Tilstandspåstand 20: En av betingelsene i løkka er false, eller begge er det. Mao. ord.neste == null eller ord.neste.ordet.equals(s) == true eller begge.

Listehode og -hale.

Når vi lager metoder (algoritmer) som leter og tar ut objekter i lenkelister, ser

vi at det generelle tilfellet er at objektet ligger i lenkelista med et objekt både foran og bak. Objektene først og sist må vi ofte behandle som spesialtilfeller for å bevare tilstandspåstandene om lista. Dette kan vi unngå ved å legge noen spesielle objekter først (og sist) i lista, slik at alle «virkelige» objekter hadde minst ett objekt både foran og bak. Et slikt spesialobjekt først i lista kaller vi et listehode. Ligger det sist kaller vi det listehale. Har vi både hode og hale i ei liste, er den tom hvis lista inneholder bare disse to spesialobjektene.

Listehode og -hale er særlig nyttig når lista skal ordnes etter en verdi (f.eks. alfabetisk). Vi sier ofte at lista er sortert (på en eller annen verdi). Da får listehodet en verdi slik at den er mindre enn alle mulige «virkelige» verdier (f.eks. verdien «AAAAAA» for navn o.l.). Tilsvarende får listehalen en verdi som er større. Slik sikres at virkelige verdier alltid ligger mellom disse listeendene, og algoritmene blir enklere og kortere.

Oppgave 6 Skriv om ordlisteklassen slik at den benytter listehode og hale. Legg disse to spesialelementene til lista i konstruktøren.

Generelle lenkelister

Hittil har vi sett på lenkelister av ordobjekter. Ved å utstyre objekter med en nestepeker, kan vi lage lenkelister av andre objekter. I forelesningene og i obligene lager vi lenkelister med personer istedet for ord. Vi skal nå se på hvordan vi går fram for å endre denne listeklassen slik at den blir mer generell, dvs. kan lenke sammen objekter av andre typer.

Vi ønsker å utvikle en beholder for objekter (av ukjent type) hvor den interne datastruktur i beholderen er ei lenkeliste. Kort sagt en generell beholder for objekter.

Dette eksemplet er nokså likt ordlisteeksemplet, men med *personer* istedet for ord:

```
class Person {
    String navn;
    Person neste;

    Person(String n) {
        navn = n;
    }

    public void skriv(){
        System.out.println(navn);
    }
}
```

Klassen som inneholder personlista (her har vi et listehode, men ingen listehale):

```
public class ListeAvPersoner {
    private Person første, siste;
    private int antall;
```

```
    ListeAvPersoner(){
        // skal etablere datastrukturen for tom liste:
        Person lh = new Person("LISTEHODE!!");
        første = lh; siste = lh;
        antall = 0;
    }

    public void settInnForan(Person nypers){
        nypers.neste = første.neste;
        første.neste = nypers;
        if (siste.neste == nypers) // nyperson er ny siste!
            siste = nypers;
        antall++;
    }

    public void settInnBak(Person inn){
        siste.neste = inn;
        siste = inn;
        antall++;
    }

    public void settInnEtter(Person denne, Person nypers) {
        nypers.neste = denne.neste;
        denne.neste = nypers;
        if (siste.neste == nypers)  // nyperson er ny siste!
            siste = nypers;
        antall++;
    }

    public Person finnPerson(String s) {
        Person p = første.neste;
        for (int i = antall; i>0; i--) {
            if (p.hentNavn().equals(s)) return p;
            else p = p.neste;
        }
        return null;
    }

    // Har fjernet metoder som ikke er vesentlig for det vi skal gjøre her
    //(metoden som skriver ut bl.a.).
}
```

Oppgave 6a Utvid klassen med en metode slik at lista som lages administreres *fifo*. Hvilken metode bør fjernes (eller ikke være public) for å sikre at det kun er mulig å sette inn og ta ut *fifo*?

Hvis vi ønsker å bruke denne klassen for å lagre ord, må alle personvariable (personpekere) i listeklassen må byttes til Ord og vi må skrive om newkallet så den har samme signatur som en konstruktør i ordklassen. Her tar tilfeldigvis begge klassene en tekststreng som parameter til konstruktøren, så det blir ingen endring der.

```
public class ListeAvOrd {

    private Ord første, siste;
    private int antall;
```

```
    ListeAvOrd(){
        // skal etablere datastrukturen for tom liste:
        Ord lh = new Ord("LISTEHODE!!");
        første = lh;     siste = lh;      antall = 0;
    }

    public void settInnBak(Ord inn){
        siste.neste = inn;
        siste = inn;
        antall++;
    }
    ..... osv.
}
```

Hvis vi skulle lagre objekter som er instanser av klasser som implementerer et grensesnitt, hva må vi gjøre da?

Den viktigste ulempen med listeklassen, er at objektene som lenkes sammen bare kan være med i ei liste av gangen, siden de bare har én nestepeker. Hvis vi laget flere nestepekere, kunne vi få til så mange lister som vi hadde pekere, men det er ingen tilfredsstillende løsning! Hvordan kan vi skrive programmet slik at vi kunne lage så mange lenkelister vi ville, som inneholdt personer f.eks., uten at vi måtte klone personobjektene for hver liste?

Vi innfører følgende lille geniale klasse:

```
    class Node {
        Node (Person p) { objektet = p; }
        Node neste;
        Person personobjektet;
    }
```

Av denne klassen kan vi lage objekter (instanser) med to egenskaper:

1. de kan lenkes sammen gjennom lenka (pekeren) Node neste
2. hvert objekt kan peke på et personobjekt

Disse objektene kaller vi knuter eller noder (fra latin *nodus* som betyr knute).

Når et personobjekt skal inngå i ei liste med slike nodeobjekter, må vi i listeklassen lage et nytt nodeobjekt og sette dette til å peke på personobjektet og sette *nodeobjektet* inn i lista:

```
class Node {
  // konstruktøren setter opp peker til personobjektet:
  Node (Person p) { personobjektet = p; }
  Node neste;
  Person personobjektet;
}

public class ListeAvPersoner {
  private Node første, siste;
  private int antall;

  ListeAvPersoner(){
    // skal etablere datastrukturen for tom liste:
    Node lh = new Node(new Person("LISTEHODE!!"));
```

```
    første = lh;          siste = lh;       antall = 0;
  }

  public void settInnBak(Person nyPerson) {
    Node inn = new Node(nyPerson);
    siste.neste = inn;
    siste = inn;
    antall++;
  }

  .....
}
```

Når et personobjekt settes inn lages det ikke en kopi av det, men en ny peker til objektet. Slik kan samme personobjekt pekes på fra «uendelig mange» lenkelister.

Spørsmålet om grensenitt ovenfor antyder at det er vanskelig å bruke den første lenkelista til å lagre objekter som er instanser av en klasse som implementerer et grensesnitt, siden et grensesnitt ikke kan ha variable og da heller ingen nestepeker. Ta som eksempel dette grensesnittet for gaver:

```
public interface Gave {
  String kategori();
    // Returnerer en tekststreng som gjør det mulig å vite hva slags gave
    // dette er, f.eks. «bok», «plate», «vin», «sko», «sjokolade», «bil»...

  String gaveId();
    // Returnerer en tekststreng som identifiserer gaven. To gaver med
    // lik kategori og gaveId er samme gave (gjenstand).
}
```

En nestepeker kan ikke legges i grensesnittet, men må evt. ligge i klassen som implementerer det. Men denne pekeren er ikke tilgjengelig med «gavebriller», når vi ikke vet annet om objektet enn at det er en gave, f.eks. gjennom en peker av type Gave.

Ved å flytte nestepekeren til nodeobjektet, forsvinner dette problemet. Her er samme klasse for å lage lenkelister av gaveobjekter:

```
class Node {
  // konstruktøren setter opp peker til gaveobjektet:

  Node (Gave g) { objektet = g; }
  Node neste;
  Gave gaveobjektet;
}

public class ListeAvGaver {
  private Node første, siste;
  private int antall;

  ListeAvGaver(){
    // skal etablere datastrukturen for tom liste:
    Node lh = new Node(new Gave("LISTEHODE!!")); // hva er galt her?
    første = lh;          siste = lh;       antall = 0;
  }
```

```
  public void settInnBak(Gave nyGave) {
    Node inn = new Node(nyGave);
    siste.neste = inn;
    siste = inn;
    antall++;
  }

  .....
}
```

Legg merke til at metodene i listeklassen nå ikke forandrer på noen egenskaper i Person eller Gave. Metodene oppretter nye nodeobjekter og tilordner verdier til disse (neste og objektet). Disse to pekervariablene brukes bare av metodene inne i listeklassen.

Siden programmet utenfor listeklassen ikke trenger å ha tilgang til nodeobjektene, er det god objektorientering å skjule denne klassen ved å gjøre den til *en indre klasse*, en klasse i klassen:

```
public class ListeAvGaver {
  private Node første, siste;
  private int antall;

  private class Node {
    Node (Gave g) { objektet = g; }
    Node neste;
    Gave gaveobjektet;
  }

  ListeAvGaver(){
    // skal etablere datastrukturen for tom liste:
    Node lh = new Node(new Gave("LISTEHODE!!")); // hva er galt her?
    første = lh;          siste = lh;       antall = 0;
  }

  public void settInnBak(Gave nyGave) {
    Node inn = new Node(nyGave);
    siste.neste = inn;
    siste = inn;
    antall++;
  }

  .....
}
```

Modifikatoren private gjør at Node er usynlig utenfor klassen (vi kan ikke deklarere en variabel av type Node utenfor klassen, med mindre vi har en annen klasse med samme navn der). Hvis klassen ikke er private, er den synlig som ListeAvGaver.Node, f.eks. kunne vi da ha laget et nodeobjekt med et gaveobjekt (f.eks. et objekt av klassen Vin gitt at Vin implementerer Gave) utenfor klassen med

```
ListeAvGaver.Node etFNO = new ListeAvGaver.Node(new Vin("kvitvin"));
```

men dette har vi liten nytte av, derfor er det god programmeringsskikk og la indre klasser være skjult.

Vi kommer ikke til å bruke indre klasser til mye mer enn dette i INF1010.

Men fortsatt er listeklassen ikke i stand til å lagre annet en objekter av bestemte typer. ListeAvGaver kan riktignok lagre gaver av flere typer, så lenge objektene er instanser av klasser (eller subklasser til klasser) som implementerer grensesnittet Gave.

Object-klassen er alle klassers mor eller superklasse. En variabel av type Object kan peke på objekter av *alle* klasser.

```
public class ListeAvObjekter {
  private Node første, siste;
  private int antall;

  private class Node {
    Node (Object obj) { objektet = obj; }
    Node neste;
    Object objektet;
  }

  ListeAvObjekter(){
    // skal etablere datastrukturen for tom liste:
    Node lh = new Node(new Object());
    første = lh;          siste = lh;       antall = 0;
  }

  public void settInnBak(Object nyttObjekt) {
    Node inn = new Node(nyttObjekt);
    siste.neste = inn;
    siste = inn;
    antall++;
  }

  public Object taUtForan() { // fjerner en node og returnerer objektet
    Node n = første.neste;
    if (n != null) {
      første.neste = n.neste;
      return n.objektet;
    }
    else return null;
  }

  .....
}
```

I denne lenkelista kan vi sette inn objekter av type Object og alle objekter av klasser som er subklasser til Object, dvs. alle objekter vi kan lage i Java!

Så hvis denne fungerer for alt mulig, hva skal vi da med alternativer? Problemet med denne beholderen er at vi ikke vet mer om objektene i den enn at de har egenskapene til Object, jf. Javas API.

Vi kan sette inn (metoden settInnBak) hva som helst. Til gjengjeld vet vi ikke hva vi får når vi tar ut (metoden taUtForan). Her må vi evt. sjekke med instanceof og typekonvertere. Hvis vi vet typen på objektet som hentes ut, kan vi typekonvertere direkte. Dette slipper vi hvis vi lar typen til objektene som inn i lista være (generisk) parameter til klassen:

```
public class LenkeListe <T> {
    private Node første, siste;
    private int antall;
```

```
    private class Node {
        Node (T t) { objektet = t; }
        Node neste;
        T objektet;
    }

    LenkeListe(){
        // skal etablere datastrukturen for tom liste:
        Node lh = new Node( null ); // listehode
        første = lh;
        siste = lh;
        antall = 0;
    }

    public void settInnForan(T t) {
        Node n = new Node(t);
        n.neste = første.neste;
        første.neste = n;
        if (siste.neste == n) // n er nytt sisteobjekt!
            siste = n;
        antall++;
    }

    public T taUtForan() {
        Node n = første.neste;
        if (n != null) {
            første.neste = n.neste;
            return n.objektet;
        }
        else return null;
    }

}
```

Her er eksempler på bruk. Først en klasse hvor vi kan lagre objekter av klassen Bil. (Egentlig alle klasser som er subklasser av Bil eller subklasser til en klasse som implementerer Bil hvis Bil er et grensesnitt).

```
LenkeListe<Bil> storgarasje = new LenkeListe<Bil>();
```

En lenkeliste med personer. Ved å lage flere slike kunne erstattet personarrayene i Person (fra obligene) med slike lenkelister av personer:

```
LenkeListe<Person> mineVenner = new LenkeListe<Person>();
LenkeListe<Person> kjenner = new LenkeListe<Person>();
LenkeListe<Person> likerIkke = new LenkeListe<Person>();
```

Og et eksempel på en beholder for gaveobjekter, jf. definisjonen ovenfor. Obs. Gave er et grensesnitt:

```
LenkeListe<Gave> gaveButikk = new LenkeListe<Gave>();
```

Hvis vi i listeklassen trenger å kjenne noen av egenskapene til objektene, er det mulig. Hvis vi vet at vi bare ønsker å lagre objekter av klasser som implementerer Gave-grensesnittet, kan vi skrive

```
public class LenkeListe <T extends Gave> {
    private Node første, siste;
```

```
    private int antall;

    private class Node {
        Node (T t) { objektet = t; }
        Node neste;
        T objektet;
    }

    LenkeListe(){
        // skal etablere datastrukturen for tom liste:
        Node lh = new Node( null ); // listehode
        første = lh;
        siste = lh;
        antall = 0;
    }

    public void settInnForan(T t) {
        Node n = new Node(t);
        n.neste = første.neste;
        første.neste = n;
        if (siste.neste == n) // n er nytt sisteobjekt!
            siste = n;
        antall++;
    }

    public T taUtForan() {
        Node n = første.neste;
        if (n != null) {
            første.neste = n.neste;
            return n.objektet;
        }
        else return null;
    }

}
```

Her kan vi få tak i gaveegenskaper (fra grensesnittet) inne i klassen, f.eks. slik:

```
public T finnGaveIKategori(String s) {
    Node n = første.neste;
    for (int i = antall; i>0; i--) {
        if (n.objektet.kategori().equals(s)) return n.objektet;
        else n = n.neste;
    }
    return null;
}
```

Den lokale variabelen n peker på en node. Noden har en variabel objektet som peker på et objekt av typen T som vi vet har gaveegenskaper, dvs. den har bl.a. en metode kategori().

I dette eksemplet har vi beholdt listehodet selv om vi ikke har hatt bruk for det i metodene vi har laget. Listehoder og -haler er nyttigst i forbindelse med ordnede (sorterte) lister og/eller når lista er dobbeltlenka.

La oss til slutt bruke dette eksemplet til å lage en fifo- og en lifo-kø. Først fifo:

```
public class FIFOliste <T extends Gave> {
    private Node første, siste;
    private int antall;

    private class Node {
        Node (T t) { objektet = t; }
        Node neste;
        T objektet;
    }

    FIFOliste(){
        første = null;
        siste = null;
        antall = 0;
    }

    public void settInn(T t) {    // setter inn bakerst
        Node n = new Node(t);
        if (siste != null) {
            siste.neste = n;
            n.neste = null;
            siste = n;
        }
        else { // lista var tom
            siste = n;
            første = n;
            n.neste = null;
        }
        antall++;
    }

    public T taUt() {    // tar ut foran
        Node n = første;
        if (n != null) {
            antall--;
            første = n.neste;
            return n.objektet;
        }
        else return null;
    }
}
```

Så lifo:

```
public class Stabel <T extends Gave> {
    private Node første;
    private int antall;

    private class Node {
        Node (T t) { objektet = t; }
        Node neste;
        T objektet;
    }

    Stabel(){
        første = null;
        siste = null;
        antall = 0;
    }
```

```
    public void settInn(T t) {    // setter inn foran
        Node n = new Node(t);
        n.neste = første;
        første = n;
        antall++;
    }

    public T taUt() {    // tar ut foran
        Node n = første;
        if (n != null) {
            antall--;
            første = n.neste;
            return n.objektet;
        }
        else return null;
    }
}
```

I denne siste LIFO-køen bruker vi ikke antall til noe. Den kan derfor fjernes. Men hvis vi skal lage flere metoder, kan den komme til nytte igjen!

Lenkeliste som generisk beholder

Lenkelister er meget godt egnet til å lage *beholdere*. Jf. Stein Gjessings notat *Enkle generiske klasser i Java*. Vi bytter ut arrayen med lenkelista. Lenkelister er en hensiktsmessig datastruktur for beholderen når vi er usikre på hvor mange objekter det er behov for å lagre fordi vi slipper å ta stilling til hvor lang arrayen skal være. Og når det ikke er behov for å ha en ordning av objektene, dvs vi kan organisere innsetting og uttaking *fifo* eller *lifo*.

Her er et forsøk på en enkel beholder som fungerer som en stabel. Legg merke til at denne listeklassen ikke har noen konstruktør.

```
class Beholder <T> {
    private Node første;
    private int ant = 0;

    private class Node { // listeelement
      Node neste;
      T objektet;

      Node(T t){
        objektet = t;
      }
    }

    public void settInn(T det) {
        Node nyNode = new Node(det);
        nyNode.neste = første;
        første = nyNode;
        ant++;
    }

    public T taUt() {
        T returverdi = null;
```

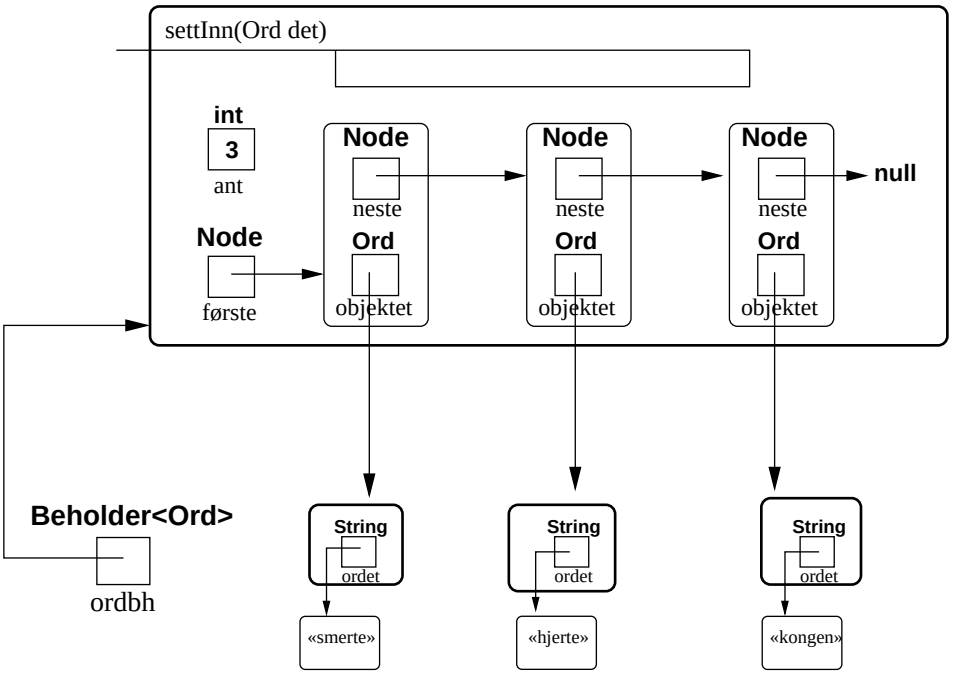
```
        if (ant > 0) {
          ant--;
          returverdi = første.objektet;
          første = første.neste;
        }

        return returverdi;
    }

     public int antall() { return ant;}
}
```

Et program for å opprette ei liste med tre ordobjekter kan da se slik ut:

```
Beholder<Ord> ordbh = new Beholder<Ord>();
ordbh.settInn(new Ord("kongen"));
ordbh.settInn(new Ord("hjerte"));
ordbh.settInn(new Ord("smerte"));
```

Lenkelista lages nå ved hjelp av nestepekeren i den indre klassen Node, så vi behøver ikke lenger nestepekerne i Ord. Vi forenkler derfor denne klassen ved å sløyfe nestepekeren.

En tegning av datastrukturen til ordbh forklarer på en grei måte hvordan beholderen vår ser ut:



Siden listeelementobjektene er av en indre klasse er disse tegnet inne i objektet ordbh peker på. I virkeligheten (i programhukommelsen til javamaskina) er det ikke slik. Beholderobjektet mangler metoden taUt og antall. Metoden settInn har et «håndtak» som stikker ut. Dette illustrerer at metoden er tilgjengelig utenfra (public). Siden ordbh peker på en beholder hvor den formelle parameteren T er erstattet med den aktuelle Ord, er alle forekomster av T byttet ut med Ord i figuren.

I denne beholderen kan vi nå lagre hva som helst, ikke bare ord. Men kan vi lage en beholder som kan inneholde objekter av forkjellige klasser? Ja, det er mulig å få til ved å benytte *grensesnitt*:

```
interface Oppbevarbar {
  public void siHvaSlagsObjektJegEr();
}
```

Dette meget enkle grensesnittet kan vi bruke til å lage pekere som kan peke på *objekter av klasser som implementerer dette grensesnittet:*

```
Oppbevarbar pekerTilOppbevarbartObjekt;
```

Gjennom denne pekeren har vi bare tilgang til et objekt som vi ikke vet noe annet om, enn at det har en siHvaSlagsObjektJegEr-metode. Her er to eksempler på klasser som denne pekeren kan peke på:

```
class Person implements Oppbevarbar {
    private String id;
    Person (String o) { id = o; }
    public void siHvaSlagsObjektJegEr(){
        System.out.println("Jeg er et personobjekt og min id er " + id);
    }
}

class Kanin implements Oppbevarbar {
    private String id;
    Kanin (String o) { id = o; }
    public void siHvaSlagsObjektJegEr(){
        System.out.println("Jeg er et kaninobjekt og min id er " + id);
    }
}
```

Oppgave 7 Skriv om beholderklassen slik at den ikke lenger er generisk og kan oppbevare objekter av klasser som implementerer grensesnittet Oppbevarbar. (Oppbevare pekere til Oppbevarbar).

I oppgaven over trenger vi ikke lenger noen parameter til klassen Beholder. Men det finnes muligheter i javaspråket til å si noe om en generisk parameter til en klasse, på samme måte som vi må si noe om typen til en metodeparameter. Dette gjøres ved å skrive om «signaturen» til klassen:

```
class Beholder <T extends Oppbevarbar> {
  // ... resten som før ...
}
```

Dette gjør at javakompilatoren sjekker at navnet vi setter inn mellom hakeparentesene når vi lager en konkret beholder, enten er grensesnittet Oppbevarbar, eller en klasse som implementerer Oppbevarbar. (Når vi har lært om subklasser brukes denne notasjonen også hvis T er en subklasse av en *klasse* Oppbevarbar eller subklasse av en klasse som implementerer Oppbevarbar. Det er antagelig grunnen til extends istedet for implements.)

Dette betyr at alle disse beholderne nå er lovlige:

```
Beholder<Oppbevarbar> skuffMedLittAvHvert;
Beholder<Person> gruppe7iINF1010;
```

```
Beholder<Kanin> mittKaninBur;
```

I den første kan vi sette inn både personer og kaniner, mens vi i de to siste bare kan oppbevare henholdsvis personer og kaniner.

Beholder som grensesnitt

Å abstrahere bort detaljer er noe vi ofte ønsker. Også når vi lager beholdere. Derfor er det vanlig at selve beholderen beskrives som et grensesnitt. Så overlates til klassen som implementerer grensesnittet å definere detaljene. Datastruktur (array, lenkeliste eller HashMap ...), køordning (FIFO, LIFO, ...) m.v. Her er et enkelt grensesnitt for en beholder:

```
interface Beholder<T> {
    public void settInn(T det);
    public T taUt();
    public int antall();
}
```

Vi kan nå lage oss «spesialbeholdere» etter behag:

```
class FifoLenkeliste<E> implements Beholder<E>
  // en generell liste administrert først inn, først ut

class LifoLenkeliste<E> implements Beholder<E>
  // en generell liste administrert sist inn, først ut
```

Trenger vi å lage en stabel av bøker, deklarerer vi:

```
LifoLenkeliste<Bok> bokstabel = new LifoLenkeliste<Bok>;
```

Forutsatt at vi har definert klassen Bok. Så kan vi legge på og ta ut bøker fra stabelen ved kallene bokstabel.settInn(Bok b) og bokstabel.taUt().

Oppgave 8 Skriv en klasse Stabel som implementerer Beholder og som kan oppbevare objekter av typen Oppbevarbar og objekter av klasser som implementerer Oppbevarbar.

Sorterte lenkelister

Lenkelister er velegnet for å holde ei liste ordnet. (Arrayer er mindre egnet fordi vi riskerer å måtte flytte objekter ofte for å opprettholde ordningen etter hvert som vi setter nye objekter inn i arrayen.)

Når vi har lagt inn mange ord i ordlista vår, vil spørsmålet om alfabetisk ordning av ordene dukke opp. Til nå har vi kun satt inn først eller sist, og det er ganske greit. Vi har også lært at å lete seg fram til et objekt i lista er mer komplisert. Det samme gjelder hvis vi ønsker at lista alltid skal være sortert ved å sette inn *på rett plass*. Å si at lista alltid er sortert betyr at vi har en *invariant tilstandspåstand* om strukturen. En påstand om datastrukturen som ikke forandrer seg. En mulighet er selvfølgelig å sette inn som før og så sortere hele lista alfabetisk etter at alt er satt inn. Å sortere en liste «etterpå» er ikke pensum i INF1010. Derimot skal vi være i stand til å sette inn objekter slik at lista alltid er sortert.

Når lista skal være sortert blir det mer komplisert å sette inn objekter. Dette fordi vi må lete oss fram til rett plass i lista der objektet skal være. Vi så tidligere at det å finne et bestemt objekt er litt mer krevende. Å ta ut objekter blir ganske likt, men kan gjøres mer effektivt fordi vi fortere kan fastslå at ordet vi leter etter ikke er der. Uten alfabetisk ordning må vi lete gjennom hele lista før vi kan fastslå det samme. Hvis vi vil forby å lagre det samme ordet flere ganger, er det også langt lettere å få til i ei sortert liste.

I INF1010 er sorterte lenkelister en viktig del av det obligatoriske arbeidet, så vi tar bare med eksempler på ordnet innsetting i det siste programeksempelet. HUSK: Man lærer dette absolutt best ved å gjøre det selv.

Men hvordan sammenligner vi to forskjellige ordobjekter? Vi så ovenfor at String-klassen inneholder metoden equals som kan avgjøre om to tekststrenger er like. Men String er en svær klasse som også har en metode for å avgjøre om en tekst alfabetisk større enn eller mindre enn en annen tekst. Hvis vi har to tekststrenger, t og s, vil kallet

```
int smlgv = t.compareTo(s)
```

gi heltallsvariabelen smlgv en verdi som vi kan inspisere. compareTo i String (se Javas API) er programmert slik at

smlgv > 0 når t $>$ s

smlgv $= 0$ når t $=$ s

smlgv < 0 når t $<$ s

Ulikhetene mellom tekstvariablene er alfabetisk, slik at t $>$ s, betyr at t er *alfabetisk større* enn s. (F.eks. er «smerte» $>$ «hjerte» fordi «h» kommer før «s» i alfabetet, «h» er mindre enn «s».)

Hvis vi nå ønsker en alfabetisk ordning av ordobjektene, innfører vi den invariante tilstandspåstanden om at hvis et ordobjekt o1 ligger foran (nærmere førstepekeren) et annet ordobjekt o2, så er o1.ordet alfabetisk mindre enn eller lik o2.ordet.

Iterere over lenkelister

Når vi har en beholder hvor datastrukturen er implementert som ei lenkeliste, er dette en velegnet struktur å iterere over. Vi itererer over lista ved å starte med første objekt og så flytte en peker til neste objekt i ei for- eller whileløkke til det ikke er flere objekter igjen. Når vi itererer over lista, må vi når vi skriver next-metoden bestemme om rekkefølgen blir fifo, lifo eller noe annet. Her er et eksempel på ei lenkeliste som har en iterator. For å få tak i definisjonene av de to grensesnittene vi bruker, må vi hente dem i biblioteket java.util.

```
import java.util.*;

public class LenkeListe <T> implements Iterable<T> {
    private Node første, siste;
    private int antall;
```

```
    private class Node {
        Node (T t) { objektet = t; }
        Node neste;
        T objektet;
    }

    LenkeListe(){
        // skal etablere datastrukturen for tom liste:
        første = null;
        siste = null;
        antall = 0;
    }

    public Iterator<T> iterator() { return new LenkeListeIterator(); }

    private class LenkeListeIterator implements Iterator<T> {
        int teller = 0;
        Node pekerTilNeste = første;

        public boolean hasNext() { return ( teller < antall );}
        public void remove() {}
        public T next() {
            T returnerDenne = pekerTilNeste.objektet;
            teller++ ;
            pekerTilNeste = pekerTilNeste.neste;
            return returnerDenne;
        }
    }

    public void settInnForan(T t) {
        Node n = new Node(t);
        n.neste = første;
        første = n;
        if (siste == null) // n er nytt sisteobjekt!
            siste = n;
        antall++;
    }

    public T taUtForan() {
        Node n = første;
        if (n != null) {
            første = n.neste;
            antall--;
            return n.objektet;
        }
        else return null;
    }

}
```

Når klassen har implementert Iterable<T> vil vi kunne skrive noe à la

```
LenkeListe<Ord> ordListe; // navnet på lista - her med Ord som parameter
                          // da er det Ord vi itererer over:

for (Ord ord : ordListe) {
 // gjør noe med ordet
 // ....
}
```

Oppsummerende eksempel

I dette siste eksemplet lager vi en listeklasse som summerer opp det vi har lært om lenkelister og noe av det vi har lært om generiske klasseparametre, grensesnitt, abstrakte klasser, subklasser, indre klasser, polymorfe metoder mm. Nytt i dette programmet er at noen av metodene «kaller på seg selv», det vi i INF1010 kaller rekursjon mellom objekter.

Vi skal lage nok en beholder for objekter som implementeres som ei lenkeliste. Objektene skal alle være av klasser som implementerer dette grensesnittet:

```
public interface Gave extends Skrivbar<Gave> {
  String kategori();
    // Returnerer en tekststreng som gjør det mulig å vite hva slags gave
    // dette er, f.eks. «bok», «plate», «vin», «sko», «sjokolade», «bil»...

  String gaveId();
    // Returnerer en tekststreng som identifiserer gaven. To gaver med
    // lik kategori og gaveId er samme gave (gjenstand).
}
```

Grensesnittet som Gave utvider er slik:

```
interface Skrivbar<E> extends Comparable<E>{
    // klasser som implementerer dette grensesnittet
    // identifiserer seg ved et kall på denne metoden

    void skriv();
}
```

Vi trenger ikke skrive Comparable<E> fordi det grensesnittet er definert i javabiblioteket. Klasser som *implementerer* Gave, må implementere disse 4 metodene:

```
String kategori()       // fra Gave
String gaveId()         // fra Gave
void skriv()            // fra Skrivbar<Gave>
int compareTo(Gave g)   // fra Comparable<Gave>
```

Eksempler på klasser som implementerer Gave. Vi lager først en abstrakt gaveklasse som implementerer metoder som er felles for alle gaver:

```
abstract class AbstrGave implements Gave {
    private String id;
    AbstrGave ( String i ) { id = i; }

    public String kategori() {return "Ubestemt kat";}
    public String gaveId() { return id; }
    public void skriv() {
        String s = kategori() + "---" + gaveId();
        System.out.print(s);
    }

    public int compareTo(Gave g) {
        return id.compareTo(g.gaveId());
    }
}
```

Eksempel på en gave. Legg merke til at det her bare er kategori-metoden som må skrives for hver konkret gave:

```
class Bok extends AbstrGave {
    private String kat, id;

    Bok ( String i ) {
        super(i);
        kat = "bok";
    }

    public String kategori() {
        return kat;
    }
}
```

Poenget med AbstrGave er at vi slipper å implementere tre av metodene mer enn en gang.

I listeklassen vet vi om objektene at de er skrivbare, dvs. at de arver metodene fra Skrivbar og dermed også fra Comparable<E>. Dette bruker vi til testing under programutviklingen (skriv()) og vi benytter compareTo() for å ordne objektene i stigende orden.

Formålet med denne siste listeklassen er å oppsummere om lenkelister og samtidig bruke mekanismer fra arv og polymorfi. Blant det som vises er hvordan vi bruker rekursjon (i nodeobjektene), i metodene skriv og settInn. Legg merke til hvordan slike metoder kommer i *par*, én offentlig i listeklassen som kan kalles fra programmet som bruker lista, og én intern i nodeklassen som bruker rekursjon. Objektene (av type Skrivbar<T>) settes inn i ordnet rekkefølge på grunnlag av verdien compareTo gir.

Eksemplet har et hierarki av noder. Grunnen til dette er at noen metoder skal virke helt likt i alle noder, mens andre metoder skal virke forskjellig i listehodet og i listehalen. Ved å la (vanlig) node og listehode og -hale være av forskjellige klasser (som alle er subklasser til Node) får vi til dette. Det opprettes bare ett objekt av klassene ListeHode og ListeHale, mens vanlige noder er instanser av klassen SKBNode. Denne klassen er det så mange objekter av som det er objekter i lista. Eksemplet viser også hvordan vi kan lage tilfeldige tall i Java.

```
public class LenkeListe<T extends Skrivbar<T>> {
    private Node førsteNode;
    private int antall;

    // rn peker til et objekt som inneholder
    // metoder for å lage tilfeldige tall
    java.util.Random rn = null;

    LenkeListe(){
        førsteNode = new ListeHode();
        førsteNode.nesteNode = new ListeHale();
        antall = 0;
        rn = new java.util.Random();
    }
```

```
private abstract class Node { // node betyr knute, fra latin nodus
    // listehodet, listehalen og alle 'ordentlige' noder er subklasser av denne
    T skrbObj = null;
    Node nesteNode = null;

    Node finnNode(T t) {
        Node returPeker = null;
        if (skrbObj != null && t.compareTo(skrbObj) == 0)
            returPeker = this;
        else if (nesteNode != null) returPeker = nesteNode.finnNode(t);
        return returPeker;
    }

    // Disse metoden må kunne kalles i alle noder, men
    // vil virke forskjellig i listehodet og -halen.
    abstract int compareTo(Node n);
    abstract void skriv();
    abstract void settInn(Node n);
}

private class ListeHode extends Node {
    // denne klassen lages det bare ett objekt av, listehodet

    int sammenligningsverdi = Integer.MIN_VALUE;

    int compareTo(Node n) {
        return sammenligningsverdi;
    }

    void skriv(){
        nesteNode.skriv(); // nulltest ikke nødvendig, hvorfor?
    }

    void settInn(Node ny) {
        if (nesteNode.compareTo(ny) > 0) {
            // objektet skal inn etter listehodet
            ny.nesteNode = nesteNode;
            nesteNode = ny;
            antall++;
        }
        else nesteNode.settInn(ny);
    }
}

private class ListeHale extends Node {
    // denne klassen lages det bare ett objekt av, listehalen
    int sammenligningsverdi = Integer.MAX_VALUE;

    int compareTo(Node n) {
        return sammenligningsverdi;
    }

    void skriv() {} // ingenting mer å skrive ut; stopper rekursjonen!

    void settInn(Node ny) {
        System.out.println("FEIL: settInn(Node ny) kalt i listehalen!");
    }
}
```

```
private class SKBNode extends Node {
    // en «ordentlig» node med peker til T

    SKBNode(T ptso) { // Peker Til Skrivbart Objekt
        skrbObj = ptso;
    }

    int compareTo(Node n) {
        return skrbObj.compareTo(n.skrbObj);
    }

    void skriv(){
        skrbObj.skriv();
        System.out.println();
        nesteNode.skriv();
    }

    void settInn(Node ny) {
        if ( nesteNode.compareTo(ny) < 0)
            nesteNode.settInn(ny);
        else if (nesteNode.compareTo(ny) > 0) {
            ny.nesteNode = nesteNode;
            nesteNode = ny;
            antall++;
        }
    }

} // class SKBNode


// Metoder i listeklassen som er tilgjengelige utenfra. De to
// siste bruker rekursjon. Legg merke til at slike rekursive metoder
// kommer i par fordi den rekusive metoden i nodene ikke er tilgjengelig
// utenom listeklassen.

public int antall() { return antall; }

T finn(T likDenne) {
    T returPeker = null;
    Node n = førsteNode.finnNode(likDenne);
    if (n != null) returPeker = n.skrbObj;
    return returPeker;
}

public T hentOgFjernTilFeldig() {
    int tilfHeltall = Math.abs(rn.nextInt()) % antall;
    // et tall mellom 0 og antall-1
    Node n = førsteNode;
    for (int i = 0 ; i < tilfHeltall; i++)
        n = n.nesteNode;
    Node nn = n.nesteNode;
    // fjerner objektet nn peker på fra lista:
    n.nesteNode = n.nesteNode.nesteNode;
    antall--;
    return nn.skrbObj;
}
```

```
    public T hentTilFeldig() {
        int tilfHeltall = Math.abs(rn.nextInt()) % antall;
        // et tall mellom 0 og antall-1
        Node n = førsteNode;
        for (int i = 0 ; i < tilfHeltall; i++)
            n = n.nesteNode;
        Node nn = n.nesteNode;
        return nn.skrbObj;
    }

    public void settInn (T nySkrivbar){

        // eneste sted i programmet hvor det lages nye noder med et skrivbart objekt

        SKBNode nyNode = new SKBNode(nySkrivbar);

        førsteNode.settInn(nyNode);
    }

    public void skrivAlle() {
        førsteNode.skriv();
    }

}
```

Dette siste eksemplet vil være tilgjengelig fra undervisningsplanen i INF1010. Her finnes det sammen med testomgivelser, klar til kompilering og uttesting.