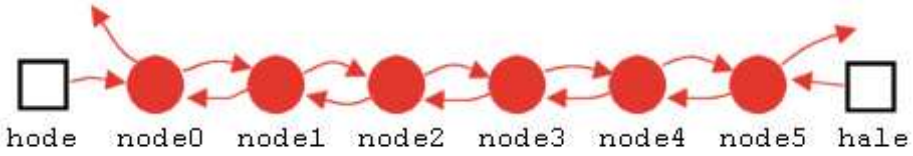
3.3 En lenket liste

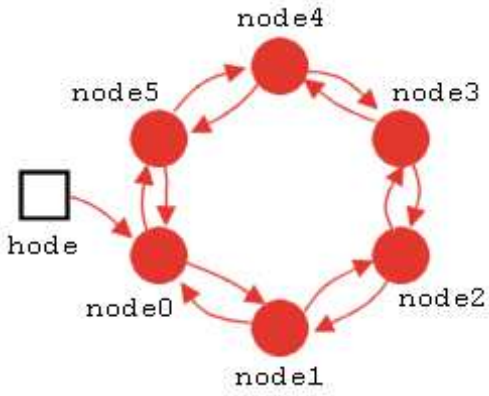
3.3.1 Lenket liste med noder

En *lenket liste* (eller en *pekerkjede* som det også kalles) består av en samling noder som er lenket sammen (i en kjede). Vi kan ha en enkeltlenket liste med hode, en dobbeltlenket liste med hode og hale, en sirkulær dobbeltlenket liste og andre varianter.

Figur 3.3.1 a) : En enkeltlenket liste med 6 noder



Figur 3.3.1 b) : En dobbeltlenket liste med 6 noder



Figur 3.3.1 c) : En dobbeltlenket sirkulær liste med 6 noder

I en enkeltlenket liste har vi en referanse (*hode*) som går til den første (indeks 0) noden i listen. I en dobbeltlenket liste har vi to referanser - en (*hode*) som går til den første i listen og en (hale) som går til den siste. I en sirkulær dobbeltlenket liste kan vi gå både forover og bakover fra node 0. En referanse kan være *null*. På figurene er det markert ved piler som ikke går til en node, men går til «ingenting».

Fordelen med en dobbeltlenket liste er at når vi skal lete etter en node med oppgitt indeks, kan vi lete forfra hvis indeks hører hjemme i første halvpart av indekser og starte letingen bakfra hvis indeks hører til andre halvpart. Det vil i gjennomsnitt halvere arbeidet i forhold til en enkeltlenket liste. I en enkeltlenket liste må vi alltid lete fra venstre ende (hodet).

Ulempen med en dobbeltlenket liste er selvfølgelig at vi må reservere plass til to referanser i hver node - en som går til neste node og en som går til forrige. Med andre ord må vi bruke dobbelt så mye minne til referanser som det vi behøver å bruke i en enkeltlenket liste.

3.3.2 En enkeltlenket liste med indre noder

Grensesnittet `Liste` ble diskutert i *Avsnitt* 3.2.1. Vi lager her en enkeltlenket liste som implementerer dette grenesnittet. Fordelen med en enkeltlenket liste i forhold til en dobbeltlenket liste er mindre minnebruk og og enklere koding.

```
public class EnkeltLenketListe<T> implements Liste<T>
{
  private static final class Node<T>       // en «nøstet» klasse
  {
    private T verdi;
    private Node<T> neste;

    private Node(T verdi, Node<T> neste)   // konstruktør
    {
      this.verdi = verdi; this.neste = neste;
    }
  }  // Node

  // her skal variabler, konstruktører og metoder komme
}
```

Programkode* 3.3.2 *a)

Node er satt opp som en generisk, privat og «nøstet» (eng: nested) klasse i listeklassen og har to private instansvariabler - en generisk verdi og en referanse til den neste noden. Node er en hjelpeklasse for listeklassen og skal dermed være privat. Den er også satt opp som statisk siden dens instanser ikke vil være avhengig av å ha kontakt med listeklassens variabler. Når en indre klasse er statisk opprettes det ingen bindinger til den ytre klassen, og dermed kan kompilatoren generere enklere kode. Det er heller ikke aktuelt å lage subklasser av Node. Dermed settes den til å være konstant (final). Det vil også kunne gi fordeler under kompilering.

Nodeklassen er satt opp med en privat konstruktør. Normalt lager man en konstruktør privat for å hindre at det blir laget intanser av klassen ved hjelp av den konstruktøren. Men i Java er det laget slik at alle variabler, metoder og konstruktører i en indre klasse, både de offentlige og de private, er åpent tilgjengelige fra den ytre klassen.

Hva slags instansvariabler bør vi ha i listeklassen? Vi må i hvert fall ha en referanse til den første noden. Hvis den er null, er listen tom. Det er også gunstig å ha en variabel som holder rede på antallet verdier i listen. Hvis ikke, måtte vi traversere listen og telle opp for å finne antallet. Et antall på 0 vil også fortelle at listen er tom.

```
private Node<T> hode;          // referanse til første node i listen
private int antall;            // antall verdier/noder i listen
```

Programkode* 3.3.2 *b)

Koden til standardkonstruktøren blir dermed slik:

```
public EnkeltLenketListe()   // standardkonstruktør
{
  hode = null;               // hode er null
  antall = 0;                // ingen verdier - listen er tom
}
```

Programkode* 3.3.2 *c)

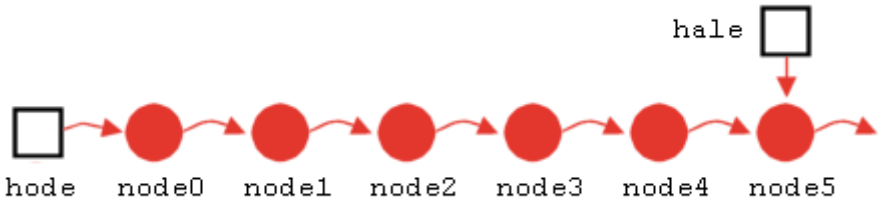
En enkeltlenket liste, med f.eks. 6 noder, tegnes vanligvis slik:

Figur 3.3.2 a) : En enkeltlenket liste med 6 noder

Det skal lite arbeid til å sette inn en ny node helt forrest i en enkeltlenket liste med et hode. Det kan gjøres ved at det 1) opprettes en ny node med gitt verdi, 2) nestereferansen i den nye noden settes til den første i listen og 3) hode settes til den nye noden. Alt dette kan imidlertid gjøres ved hjelp av kun én programsetning:

```
hode = new Node<>(verdi, hode);     // hode refererer til første node
```

Men generelt gjelder at hvis en ny node skal inn på en bestemt plass i listen, må vi starte ved hodet og lete oss frem til denne plassen. Dermed øker letearbeidet jo lenger ut i listen vi skal. Grensesnittet `Liste` krever at metoden *`leggInn`*`(T verdi)` skal legge *`verdi`* bakerst. Hvis vi lar listen også ha en *`hale`*, dvs. en referanse til den siste noden, får vi direkte aksess til den siste noden. Da vil *`leggInn`*`(T verdi)` kunne bli effektiv. Det kan tegnes slik:



Figur 3.3.2 b) : En enkeltlenket liste med 6 noder og halepeker

Med en *`hale`* i tillegg må også konstruktøren endres:

```
private Node<T> hode, hale;   // referanser til første og siste node
private int antall;           // antall verdier/noder i listen

public EnkeltLenketListe()   // standardkonstruktør
{
  hode = hale = null;        // hode og hale til null
  antall = 0;                // ingen verdier - tom liste
}
```

Programkode 3.3.2 d)

Metoden *`leggInn`*`(T verdi)` skal legge inn *`verdi`* bakerst i listen. Hvis den i utgangspunktet er tom, vil både *`hode`* og *`hale`* være null. Det er situasjonen til venstre i figuren under. En innlegging foregår da ved at både *`hode`* og *`hale`* settes til å peke på en ny node (med *`verdi`*). Referansen *`neste`* i den nye noden skal være null siden dette er siste node. Dermed får vi situasjonen til høyre i figuren under:

Figur 3.3.2 c) : Før og etter innlegging i en tom liste

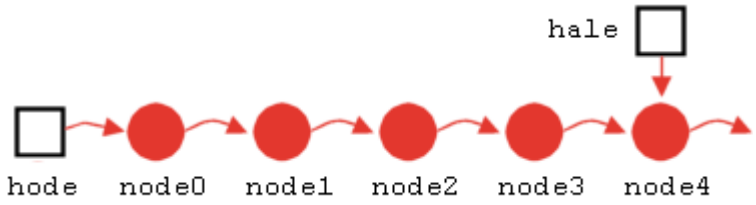
Det som skjer i *Figur* 3.3.2 *c)*, kan utføres ved to setninger, men like gjerne med kun én:

```
hale = new Node<>(verdi, null);      // bruker nodekonstruktøren
hode = hale;                         // hode får ny verdi
```

eller

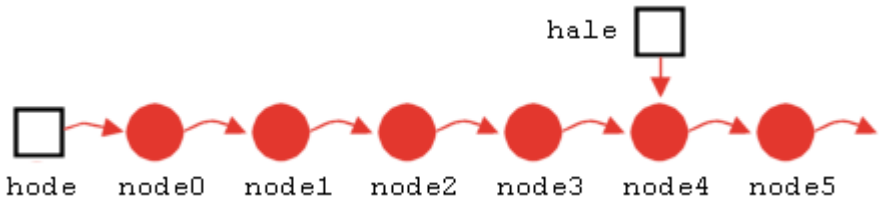
```
hode = hale = new Node<>(verdi, null);    // to setninger i ett
```

Normalt vil en liste ikke være tom. I figuren under er det fem noder – fra node0 til node4:

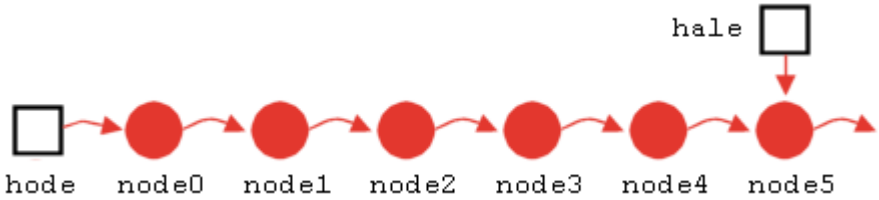


Figur 3.3.2 d) : En liste med fem noder/verdier

En innlegging bakerst skjer ved at referansen *neste* i den siste noden (*hale.neste*) settes til en ny node, og deretter flyttes halen til den nye noden:



Figur 3.3.2 e) : Ny node bakerst: hode.neste = new Node<>(verdi,null);



Figur 3.3.2 f) : «Halen» flyttes til den bakerste noden: hale = hale.neste;

Det som skjer i *Figur* 3.3.2 *e)* og *Figur* 3.3.2 *f)* kan uføres ved hjelp av kun én setning:

```
hale = hale.neste = new Node<>(verdi, null);    // ny verdi legges bakerst
```

Programkode* 3.3.2 *e)

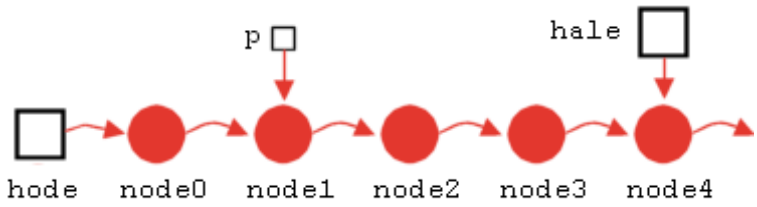
Beskrivelsene og kodebitene over, kan settes sammen til flg. kode for *leggInn*(T verdi):

```
public boolean leggInn(T verdi)   // verdi legges bakerst
{
  Objects.requireNonNull(verdi, "Ikke tillatt med null-verdier!");

  if (antall == 0)  hode = hale = new Node<>(verdi, null);  // tom liste
  else hale = hale.neste = new Node<>(verdi, null);         // legges bakerst

  antall++;                  // en mer i listen
  return true;               // vellykket innlegging
}
```

Programkode* 3.3.2 *f)

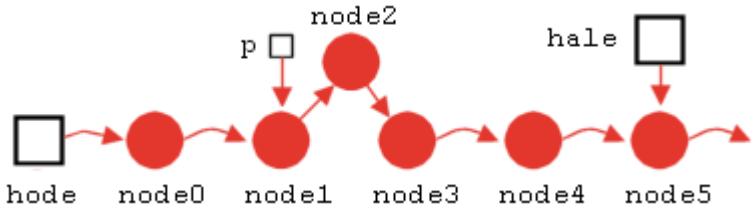
I innleggingsmetoden *leggInn(int indeks, T verdi)* skal *verdi* legges slik at posisjonen blir lik *indeks*. Det betyr at hvis *indeks* er lik 0, skal *verdi* legges først. Hvis *indeks* er lik 1, skal den legges nest først, osv. Hvis *indeks* er lik *antall*, skal den legges bakerst. Hvis *indeks* er negativ eller er større enn *antall*, er den ulovlig. Hvis *indeks* er lovlig og ulik både 0 og *antall*, må vi finne noden rett foran der den nye skal inn. Det gjøres i en løkke der en hjelpevariabel (en nodereferanse) *p* starter på første node og flyttes *indeks* - 1 ganger.

Figuren under viser at *p* må stoppe på *node*1 hvis ny verdi skal inn på plass/indeks 2:



Figur 3.3.2 g) : Ny verdi skal inn på plass/indeks 2

Den nye noden legges mellom *node*1 og *node*2. Etter innleggingen blir den nye noden *node*2 og de som kommer etter får indekser som er én mer en de hadde:



Figur 3.3.2 h) : Den nye noden ligger nå på plass/indeks 2

```
public void leggInn(int indeks, T verdi)    // verdi til posisjon indeks
{
  Objects.requireNonNull(verdi, "Ikke tillatt med null-verdier!");

  indeksKontroll(indeks, true);  // Se Liste, true: indeks = antall er lovlig

  if (indeks == 0)                      // ny verdi skal ligge først
  {
    hode = new Node<>(verdi, hode);     // legges først
    if (antall == 0) hale = hode;       // hode og hale går til samme node
  }
  else if (indeks == antall)            // ny verdi skal ligge bakerst
  {
    hale = hale.neste = new Node<>(verdi, null);  // legges bakerst
  }
  else
  {
    Node<T> p = hode;                   // p flyttes indeks - 1 ganger
    for (int i = 1; i < indeks; i++) p = p.neste;

    p.neste = new Node<>(verdi, p.neste);  // verdi settes inn i listen
  }

  antall++;                             // listen har fått en ny verdi
}
```

Programkode 3.3.2 g)

Oppgaver til Avsnitt 3.3.2

1. `EnkeltLenketListe` inneholder den koden som til nå er diskutert i dette avsnittet. De metodene som mangler er satt opp med tom kode. Flytt dette over til deg f.eks. under mappen (package) hjelpeklasser.
2. Lag metodene *`antall`*`()`, *`tom`*`()`, *`nullstill`*`()` og *`toString`*`()` i `EnkeltLenketListe`. Se *Oppgave* 1. Metoden *`antall`*`()` skal returnere antallet verdier i listen, *`tom`*`()` skal returnere sann/true hvis listen er tom (antall er 0) og returnere usann/false ellers og *`nullstill`*`()` skal «tømme» listen slik at den blir tom. Metoden *`toString`*`()` skal lages slik at den returnerer [] hvis listen er tom, [3] hvis den kun inneholder verdien 3 og [1, 2, 3] hvis den f.eks. inneholder 1, 2 og 3.
3. Lag konstruktøren `public EnkeltLenketListe(T[] a)`. Den skal gjøre at verdiene får samme rekkefølge i listen som de har i tabellen. Dette skal kodes direkte uten bruk av noen av *`leggInn`*-metodene.
4. Lag et program (utenfor klassen `EnkeltLenketListe`) der det opprettes en instans av klassen med `Integer` som typeparameter. Legg så inn en del verdier (heltall). Bruk begge *`leggInn`*-metodene. Sjekk så ved å lage utskrifter at metodene *`antall`*`()` og *`tostring`*`()` gir rett svar. Se *Oppgave* 2. Bruk også metoden *`nullstill`*`()`.

3.3.3 Klassens øvrige metoder

I metodene *`hent()`*, *`oppdater()`* og *`fjern(indeks)`* er det nødvendig å finne noden som har en gitt posisjon/indeks. Den finner vi ved å la en referansevariabel *p* starte i listens hode og flytte seg videre *`indeks`* antall ganger. Vi lager en hjelpemetode som gjør dette for oss:

```
private Node<T> finnNode(int indeks)
{
  Node<T> p = hode;
  for (int i = 0; i < indeks; i++) p = p.neste;
  return p;
}
```

Programkode 3.3.3 a)

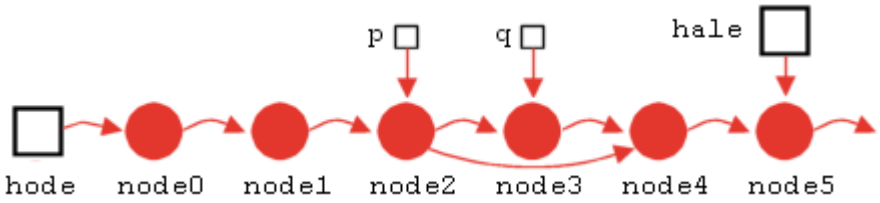
Metodene *`hent`*`()` og *`oppdater`*`()` kan nå enkelt kodes ved hjelp av metoden *`finnNode`*`()`:

```
public T hent(int indeks)
{
  indeksKontroll(indeks, false);  // Se Liste, false: indeks = antall er ulovlig
  return finnNode(indeks).verdi;
}

public T oppdater(int indeks, T verdi)
{
  Objects.requireNonNull(verdi, "Ikke tillatt med null-verdier!");

  indeksKontroll(indeks, false);  // Se Liste, false: indeks = antall er ulovlig

  Node<T> p = finnNode(indeks);
  T gammelVerdi = p.verdi;

  p.verdi = verdi;
  return gammelVerdi;
}
```

Programkode 3.3.3 b)

Metoden *fjern*(*int indeks*) er litt mer komplisert. Hvis den første noden skal fjernes, dvs. *indeks* = 0, flytter vi rett og slett *hode* til den neste noden. Hvis vi skal fjerne en node lenger ut i listen, må vi først finne noden som ligger **rett foran** den som skal fjernes.

Anta at det er *node*3 som skal fjernes. Da flyttes *p* til noden rett foran og en hjelpvariabel *q* settes til den noden som skal fjernes. Se figuren under:



Figur 3.3.3 a) : Nestepeker i *p* dirigeres forbi *q*

Noden *q* (*node*3) forsvinner fra listen når *p.neste* dirigeres forbi *q* (se figuren). Det gjøres ved setningen: `p.neste = q.neste;` Vi må passe på spesialtifellene. Det er når listen kun har én verdi og når den siste skal fjernes. I begge tilfeller må *hale* oppdateres:

```
public T fjern(int indeks)
{
  indeksKontroll(indeks, false);  // Se Liste, false: indeks = antall er ulovlig

  T temp;                                 // hjelpevariabel

  if (indeks == 0)                        // skal første verdi fjernes?
  {
    temp = hode.verdi;                    // tar vare på verdien som skal fjernes
    hode = hode.neste;                    // hode flyttes til neste node
    if (antall == 1) hale = null;         // det var kun en verdi i listen
  }
  else
  {
    Node<T> p = finnNode(indeks - 1);     // p er noden foran den som skal fjernes
    Node<T> q = p.neste;                  // q skal fjernes
    temp = q.verdi;                       // tar vare på verdien som skal fjernes

    if (q == hale) hale = p;              // q er siste node
    p.neste = q.neste;                    // "hopper over" q
  }

  antall--;                               // reduserer antallet
  return temp;                            // returner fjernet verdi
}
```

Programkode* 3.3.3 *c)

Oppgaver til Avsnitt 3.3.3

1. Sjekk nøye at metoden *fjern*(*int indeks*) i *Programkode* 3.3.3 *c)* oppfører seg korrekt. Sjekk spesielt at *hale* får korrekt verdi hvis den bakerste noden fjernes, *hode* får korrekt verdi hvis den første noden fjernes og at både *hode* og *hale* får korrekte verdier hvis fjerningen fører til at listen blir tom. Lag tegninger!

2. Lag metodene *indeksTil()* og *inneholder()*. Den første skal returnere indeksen til parameterverdien *verdi*. Hvis den ikke finnes i listen skal den returnere -1. Du må bruke metoden *equals* for å avgjøre om to verdier er like eller ulike. Metoden *inneholder*() skal returnere sann (true) hvis listen inneholder *verdi* og returnere usann (false) ellers. Den kan kodes ved hjelp av *indeksTil*().
3. Lag metoden public *fjern(T verdi)*. Den skal returnere sann (true) hvis et eksemplar av t har blitt fjernet og returnere usann (false) hvis *verdi* ikke finnes i listen. Her må en passe på at det blir korrekt for spesieltilfellene, dvs. at den første eller siste fjernes eller at listen blir tom etter fjerningen. Husk også at skal en verdi som ikke ligger først, fjernes, må vi ha tak i noden rett foran den som skal fjernes.
4. EnkeltLenketListe inneholder ferdig kode for de metodene som er diskutert i *Avsnitt* 3.3.3 og de som er satt opp som oppgaver. Lag et program der metodene brukes.

3.3.4 En indre iteratorklasse

Iteratorklassen EnkeltLenketListeIterator skal ligge som en privat indre klasse i EnkeltLenketListe. Internt i klassen trenger vi en referansevariabel *p* som kan flyttes fortløpende fra en node og til dens neste. Videre må vi ha en variabel som avgjør om det er tillatt å kalle *remove*() eller ikke. Se *Tabell* 3.1.1.

```
private class EnkeltLenketListeIterator implements Iterator<T>
{
  private Node<T> p = hode;         // p starter på den første i listen
  private boolean fjernOK = false;  // blir sann når next() kalles

  // metodene hasNext(), next() og remove() skal inn her

} // class EnkeltLenketListeIterator
```

Programkode* 3.3.4 *a)

Metoden hasNext() skal sjekke om *p* har gått ut av listen eller ikke. Den kan kodes slik:

```
public boolean hasNext()
{
  return p != null;  // p er ute av listen hvis den har blitt null
}
```

Programkode* 3.3.4 *b)

Metoden next() skal returnere «denne» (eng: current) verdien, dvs. verdien i noden *p* og samtidig flytte *p* til neste node eller til null hvis *p* var den siste. Hvis *next*() kalles med *p* ute av listen, skal det kastes en *NoSuchElementException*. Dette kan lages slik:

```
public T next()
{
  if (!hasNext()) throw new NoSuchElementException("Ingen verdier!");

  fjernOK = true;             // nå kan remove() kalles
  T denneVerdi = p.verdi;     // tar vare på verdien i p
  p = p.neste;                // flytter p til den neste noden

  return denneVerdi;          // returnerer verdien
}
```

Programkode* 3.3.4 *c)

Metoden *remove()* er satt opp som en *default*-metode i grensesnittet `Iterator`. Men den er kun kodet med en `UnsupportedOperationException`. Vi skal kode den her og det litt krevende. Det er mye å passe på. Den skal fjerne verdien som siste kall på *next*() returnerte. Problemet er at i metoden *next*() flyttes *p*. Dermed er det forgjengeren til *p* som skal fjernes. Og skal den fjernes, må vi også ha tilgang til dens forgjenger, dvs. forgjengeren til forgjengeren til *p*. Dette kan løses ved å ha flere referansevariabler i iteratorklassen.

I en iterator er det imidlertid metoden *next*() som vanligvis brukes mest, mens *remove*() brukes sjeldnere. Dermed kan det være gunstig å ikke gjøre *next*() mindre effektiv enn den er nå. Når vi i *remove*() trenger forgjengeren til forgjengeren til *p*, kan det isteden løses ved at vi leter etter den ved hjelp av en løkke. Teknikken med å ha en ekstra nodereferanse i tillegg til *p* i iteratorklassen, tas opp i *Oppgave* 6.

```
public void remove()
{
  if (!fjernOK) throw new IllegalStateException("Ulovlig tilstand!");

  fjernOK = false;                // remove() kan ikke kalles på nytt
  Node<T> q = hode;               // hjelpevariabel

  if (hode.neste == p)            // skal den første fjernes?
  {
    hode = hode.neste;            // den første fjernes
    if (p == null) hale = null;   // dette var den eneste noden
  }
  else
  {
    Node<T> r = hode;             // må finne forgjengeren
                                  // til forgjengeren til p
    while (r.neste.neste != p)
    {
      r = r.neste;                // flytter r
    }

    q = r.neste;                  // det er q som skal fjernes
    r.neste = p;                  // "hopper" over q
    if (p == null) hale = r;      // q var den siste
  }

  q.verdi = null;                 // nuller verdien i noden
  q.neste = null;                 // nuller nestereferansen

  antall--;                       // en node mindre i listen
}
```

Programkode 3.3.4 d)

Nå er klassen `EnkeltLenketListeIterator` ferdig. Det som imidlertid står igjen i er å kode metoden *iterator*(). Den skal rett og slett returnere en instans av iteratorklassen:

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

Programkode 3.3.4 e)

I *Avsnitt* 3.2.5 ble det diskutert hvordan en skal takle det at det skjer endringer i en liste etter at en iterator har startet. Iteratorens *remove*-metode fjerner en verdi som iteratoren allerede har vært innom. Men hvis det utføres en endring i en av listens mutator-metoder, vil iteratoren kunne gi uventede verdier. Det samme skjer hvis det er startet flere iteratorer og remove i en av de andre brukes.

Dette kan vi behandle på samme måte som i *Avsnitt* 3.2.5, ved hjelp av to tellere: *endringer* og *iteratorendringer*. Den første hører til listen og den andre til iteratoren. Dermed må vi ha flg. variabler i EnkeltLenketListe:

```
private Node<T> hode, hale;   // referanser til første og siste node
private int antall;           // antall verdier/noder i listen
private int endringer;        // endringer i listen
```

Programkode 3.3.4 f)

Konstruktøren må se slik ut:

```
public EnkeltLenketListe()   // standardkonstruktør
{
  hode = hale = null;        // hode og hale til null
  antall = 0;                // ingen verdier - listen er tom
  endringer = 0;             // ingen endringer når vi starter
}
```

Programkode 3.3.4 g)

Videre må vi ha med setningen *endringer*++; i alle mutatorene, dvs. i metodene for innlegging, fjerning, oppdatering og nullstilling.

Iteratoren må ha en egen endringsvariabel. Den starter med den verdien *endringer* har. Begge oppdateres hver gang iteratorens *remove*-metode brukes. Hvis disse så blir forskjellige, skyldes det en endring som er foretatt et annet sted. Iteratorklassen skal ha disse variablene:

```
private Node<T> p = hode;          // p starter på den første i listen
private boolean fjernOK = false;   // blir sann når next() kalles
private int iteratorendringer = endringer;  // startverdi
```

Programkode 3.3.4 h)

De to endringsvariablene må sammenlignes både i *next*() og i *remove*(). Hvis de er ulike, kastes en ConcurrentModificationException. Se Oppgave 1.

Oppgaver til Avsnitt 3.3.4

1. EnkeltLenketListe inneholder ferdig kode for hele klassen (inklusiv iteratoren). Lag et program der iteratoren brukes. Sjekk at det blir en ConcurrentModificationException hvis det skjer en endring etter at en iterator har startet.
2. Sjekk at de arvede metodene *fjernHvis*() og *forEach*() virker.
3. Metoden *fjernHvis*() arves fra Beholder. Kod den direkte uten å bruke iteratoren.
4. Klassen arver metoden *forEach*() fra Iterable. Kod den uten å bruke iteratoren.
5. Kod metoden *forEachRemaining*() direkte i EnkeltLenketListeIterator.
6. La klassen EnkeltLenketListeIterator ha to variabler *p* og *q*. I metoden *next*() skal normalt *q* være forgjengeren til forgjengeren til *p*, dvs. at q.neste.neste == p. Da kan *remove*() kodes mer effektivt. Her må en imidlertid passe på noen spesialtilfeller.

7. Lag klassen `DobbeltLenketListe<T>`. Den skal implementere `Liste<T>`. Nodeklassen skal ha to pekere - *neste* og *forrige*.

3.3.5 Hvilken listeimplementasjon er «best»?

Vi kan gjøre flg. sammenligninger mellom de to listeimplementasjonene, dvs. `TabellListe` og `EnkeltLenketListe`:

- `TabellListe` bruker tilsynelatende kun halvparten av plassen til `EnkeltLenketListe`. Det kommer av at hver node i `EnkeltLenketListe` har en nestereferanse i tillegg til en verdi. Men der brukes nøyaktig den plassen som trengs, mens den interne tabellen i `TabellListe` normalt lages en del større enn det som trengs i øyeblikket. Det betyr at de to likevel ikke adskiller seg så veldig mye med hensyn på plassbehov.
- Søking og oppdatering (metodene *hent* og *oppdater*) er av konstant orden i `TabellListe` og av orden *n* (der *n* er antallet verdier i listen) i `EnkeltLenketListe`.
- Det å legge en verdi bakerst i listen er av konstant orden for både `TabellListe` og for `EnkeltLenketListe`.
- Det å legge en verdi forrest i listen er av konstant orden i `EnkeltLenketListe`, men av orden *n* i `TabellListe`. Det kommer av at i `TabellListe` må alle verdiene flyttes før det kan legges noe forrest i tabellen.
- Det å legge en verdi på et vilkårlig sted i listen er av orden *n* for både `TabellListe` og for `EnkeltLenketListe`. I `TabellListe` er det å finne plassen av konstant orden, men det å flytte på verdier slik at plassen blir ledig, er av orden *n*. I `EnkeltLenketListe` er det å finne plassen av orden *n*, men det å flytte på verdier slik at plassen blir ledig, er av konstant orden.
- Metodene *indeksTil*, *inneholder* og *fjern* er av orden *n* både i `TabellListe` og i `EnkeltLenketListe`. I `TabellListe` er det å finne den som skal slettes av konstant orden, men det å «tette» igjen «hullet» i tabellen er av orden *n*. I `EnkeltLenketListe` er det å finne verdien av orden *n*, men det å ta vekk en node er av konstant orden.

Konklusjonen er avhengig av hvilke operasjoner som skal brukes mest. Hvis f.eks. hver verdi som skal legges inn, skal legges foran de som allerede er der, så vil *EnkeltLenketListe* være best. Hvis derimot hver verdi som skal legges inn, skal legges bak de som allerede er der, så vil begge være like gode. Hvis det ofte er behov for å hente eller å oppdatere verdier på bestemte plasser, så er `TabellListe` best. Dette betyr at normalt vil en `TabellListe` (eller en `ArrayList` fra `java.util`) være det beste valget.

3.3.6 Klassen LinkedList i java.util

Klassen `LinkedList` har en dobbelt lenket liste med hode og hale som intern datastruktur, dvs. slik som *Figur 3.3.1 b)*. Klassen er deklarert slik:

```
public class LinkedList<E>
  extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
```

Klassen implementerer `List` og har dermed metodene i *Programkode 3.2.1 b)* og enda flere (se *List*). I tilegg implementeres `Deque`. Navnet er konstruert ved hjelp av bokstaver fra **D**ouble **e**nded **qu**eue, dvs. en en toveiskø. `Deque` er deklarert slik:

```
public interface Deque<E> extends Queue<E>
```

Dette betyr at klassen `LinkedList` kan brukes som en vanlig kø. Begrepet kø tas opp *Delkapittel 4.2*. Flg. kø-metoder er derfor tilgjengelige:

```
boolean add(E e)     // legger inn bakerst i køen
boolean add(E e)     // legger inn bakerst i køen
boolean offer(E e)   // legger inn bakerst i køen
E remove()           // tar ut den første i køen
E poll()             // tar ut den første i køen
E element()          // ser på den første i køen
E peek()             // ser på den første i køen
```

Det er to av hver type. Både `remove` og `poll` ta ut den første i køen. `remove` returnerer `null` hvis køen er tom, mens `poll` kaster et unntak.

En toveiskø (en Deque) må ha metoder som kan arbeide i begge ender av en kø - både forrest og bakerst. Dette tas opp i *Delkapittel 4.3*. Derfor har klassen `LinkedList` også disse metodene:

```
void addFirst(E e)        // legger inn først i køen
void addLast(E e)         // legger inn bakerst i køen
boolean offerFirst(E e)   // legger inn først i køen
boolean offerLast(E e)    // legger inn bakerst i køen
E removeFirst()           // tar ut den første i køen
E removeLast()            // tar ut den siste i køen
E pollFirst()             // tar ut den første i køen
E pollLast()              // tar ut den siste i køen
E getFirst()              // ser på den første i køen
E getLast()               // ser på den siste i køen
E peekFirst()             // ser på den første i køen
E peekLast()              // ser på den siste i køen
```

Klassen `LinkedList` kan også brukes som en stakk (dette tas opp i *Delkapittel 4.1*). Flg. metoder i `LinkedList` er stakk-metoder:

```
void push(E e)   // legger inn øverst på stakken
E pop()          // tar ut øverst fra stakken
E peek()         // ser på den øverste på stakken
```

Copyright © Ulf Uttersrud, 2018. All rights reserved.