Algoritmer og datastrukturer
Kapittel 3 – Delkapittel 3.2

3.2 En tabellbasert liste

3.2.1 Grensesnittet Liste

En resultatliste har en rekkefølge.

Ordet *liste* brukes som navn på ulike begreper i vårt fag. Vi oppfatter vanligvis en *liste* som en bestemt rekkefølge av ting. Det kan for eksempel være en prisliste, en resultatliste, en bokliste, osv. I *Delkapittel 3.1* ble begrepet *beholder* diskutert. Her bestemmer vi at en *liste* rett og slett skal være en *beholder* der verdiene ligger i en bestemt rekkefølge. Med andre ord skal det være mulig både å legge inn og å spørre etter en verdi på en bestemt plass (posisjon eller indeks) i en liste.

Vi lager derfor et grensesnitt `Liste` som arver `Beholder` og som har metoder som gjør at verdiene i en liste får en bestemt rekkefølge. I en liste fra hverdagen, f.eks. en resultatliste, er det vanlig å si at den første i listen er nr. 1. Men her skal vi ha det på samme måte som i en tabell, dvs. at den første i en liste er listens element nr. 0 (eller om vi vil har posisjon/indeks lik 0).

```
public interface Liste<T> extends Beholder<T>
{
  public boolean leggInn(T verdi);            // Nytt element bakerst
  public void leggInn(int indeks, T verdi);   // Nytt element på plass indeks
  public boolean inneholder(T verdi);         // Er verdi i listen?
  public T hent(int indeks);                  // Hent element på plass indeks
  public int indeksTil(T verdi);              // Hvor ligger verdi?
  public T oppdater(int indeks, T verdi);     // Oppdater på plass indeks
  public boolean fjern(T verdi);              // Fjern objektet verdi
  public T fjern(int indeks);                 // Fjern elementet på plass indeks
  public int antall();                        // Antallet i listen
  public boolean tom();                       // Er listen tom?
  public void nullstill();                    // Listen nullstilles (og tømmes)
  public Iterator<T> iterator();              // En iterator

  public default String melding(int indeks)   // Unntaksmelding
  {
    return "Indeks: " + indeks + ", Antall: " + antall();
  }

  public default void indeksKontroll(int indeks, boolean leggInn)
  {
    if (indeks < 0 ? true : (leggInn ? indeks > antall() : indeks >= antall()))
      throw new IndexOutOfBoundsException(melding(indeks));
  }
}  // Liste
```

Programkode 3.2.1 a)

I grensesnittet *Liste* er alle metodene fra `Beholder`, bortsett fra `fjernHvis()`, blitt satt opp på nytt. Det er egentlig unødvendig siden de arves. Men det er letter å se hva *Liste* inneholder hvis de settes opp. De fem nye metodene er markert med fargen oransje. Vi ser at alle har noe med elementenes posisjon eller indeks i listen å gjøre. I tillegg er det et ekstra krav til metoden `leggInn(T t)`. Det er at `t` skal legges bakerst, dvs. som verdi i tillegg til de listen allerede har og dermed som ny siste verdi. Legg merke til at hvis `antall` er antallet elementer i en liste, så er `antall – 1` siste lovlige posisjon (indeks).

Metodene i en liste skal oppfylle flg. krav:

- Metoden *`leggInn(T verdi)`* skal legge inn verdien `verdi` bakerst i listen.
- Metoden *`leggInn(int indeks, T verdi)`* har to argumenter. Den første bestemmer hvor *`verdi`* skal legges. Det betyr at de opprinnelige verdiene i listen fra og med posisjon `indeks` og utover vil få sine posisjoner økt med 1. Vi kan si at de forskyves mot høyre (eller nedover om vi vil) i listen. Hvis `indeks` er negativ eller er større enn antallet verdier, skal det kastes en *IndexOutOfBoundsException*. Det er imidlertid lovlig med *`indeks`* lik antallet verdier. Det betyr at den nye verdien skal legges bakerst.
- Metoden *`inneholder(T verdi)`* skal gi sann hvis den inneholder *`verdi`* og usann ellers.
- Metoden *`hent(int indeks)`* skal hente (uten å fjerne) verdien på *`indeks`*. Hvis den er negativ eller >= antallet i listen, skal det kastes en *IndexOutOfBoundsException*.
- Metoden *`indeksTil(T verdi)`* skal returner plassen/indeksen til første forekomst av *`verdi`*. Hvis den ikke finnes i listen returneres −1.
- Metoden *`oppdater(int indeks, T verdi)`* skal erstatte eksisterende verdi på *`indeks`* med verdien *`verdi`*. Den gamle verdien skal returneres. Hvis det er en ulovlig indeks (negativ eller >= antallet i listen) skal det kastes en *IndexOutOfBoundsException*.
- Metoden *`fjern(int indeks)`* skal fjerne (og returnere) verdien med denne indeksen. Antallet verdier blir dermed 1 mindre enn før, og alle verdiene som kommer etterpå får redusert sin indeks med 1. Hvis det er en ulovlig indeks (negativ eller >= antallet i listen) skal det kastes en *IndexOutOfBoundsException*.
- Metoden *`fjern(T verdi)`* skal fjerne første forekomst av *`verdi`* og returnere sann. Antallet verdier blir dermed 1 mindre enn før, og alle verdiene som kommer etterpå får redusert sin indeks med 1. Hvis *`verdi`* ikke er i listen, returneres usann.
- Metoden *`antall()`* skal returnere antallet verdier og dermed 0 hvis listen er tom.
- Metoden *`tom()`* skal returnere sann (true) hvis listen er tom og usann (false) ellers.
- Metoden *`nullstill()`* skal «tømme» listen, det vil si sørge for at det som eventuelt ligger igjen blir «resirkulert»og at listen deretter blir tom.
- Metoden *`iterator()`* skal returnere en iterator for listen.
- Default-metoden *`indeksKontroll(int, boolean)`* kan brukes av alle subklasser. Metoden *`leggInn(indeks,T)`* kan legge verdien bakerst, dvs. det er tillatt med *`indeks`* lik antall verdier i listen. Da brukes *`true`* som parameter i *`indeksKontroll()`*. Hvis det ikke er tillatt med *`indeks`* lik antall verdier, brukes *`false`*.

Vi har flere muligheter hvis vi skal implementere en liste:

1) Vi kan lage en konkret klasse med en eller annen intern datastruktur og så kode metodene mest mulig effektivt med hensyn på den gitte strukturen.

2) Vi kan lage en abstrakt klasse der `leggInn(int indeks, T t)` og `iterator()` er abstrakte. De andre kan så kodes ved hjelp av dem. Det er mulig selv om mange av dem da

vil bli lite effektive. Så lages en konkret subklasse. I den holder det å kode de to abstrakte metodene. Eventuelt kan en i tillegg kode de mest ineffektive av de metodene som arves.

I neste avsnitt skal vi gå videre med idéen fra punkt 1) over, mens idéen i punkt 2) diskuteres nærmere i *Oppgave 1 og 2*.

Java Vårt grensesnitt Liste er en fornorsket miniversjon av grensesnittet *List*. Det ligger i klassebiblioteket *java.util*. I noen av metodene inngår objekter (instanser av class Object) som parameterverdier. Det hadde vært mer naturlig å bruke den generiske datatypen *T*. Men på grunn av bakoverkompabilitet er det brukt *Object*:

```
public interface List<T> extends Collection<T>
{
  public boolean add(T element);              // leggInn
  public void add(int index, T element);      // leggInn
  public boolean contains(Object o);          // inneholder
  public T get(int index);                    // hent
  public int indexOf(Object o);               // indeksTil
  public T set(int index, T element);         // oppdater
  public T remove(int index);                 // fjern
  public boolean remove(Object o);            // fjern
  public int size();                          // antall
  public boolean isEmpty();                   // tom
  public void clear();                        // nullstill
  public Iterator<T> iterator();              // iterator

  // samt mange andre metoder
}
```

Programkode* 3.2.1 *b)

Oppgaver til Avsnitt 3.2.1

1. Legg grensesnittet `Liste` under biblioteket (package) hjelpeklasser.
2. Lag den generisk abstrakte klassen `AbstraktListe`. Den skal implementere `Liste` og arve `AbstraktBeholder` (se *Avsnitt* 3.1.2). Dermed holder det å kode de metodene som er i `Liste`, men som ikke er i `Beholder`. Dvs:

   - Metoden `leggInn(int indeks, T t)` settes til å være abstrakt.
   - Metoden `leggInn(T t)` kan kodes ved hjelp av den over.
   - Lag hjelpemetoden `void indeksKontroll(int indeks)`. Den skal sjekke om indeks er lovlig, dvs. større enn eller lik 0 og mindre enn antall().
   - Metoden `hent(int indeks)` kan kodes ved hjelp av iteratoren.
   - Metoden `indeksTil(T t)` kan kodes ved hjelp av iteratoren.
   - Metoden `fjern(int indeks)` kan kodes ved hjelp av iteratoren.
   - Metoden `oppdater(int indeks, T t)` kan kodes ved hjelp av `fjern` og `leggInn`.

3. Lag klassen `EnkelTabellListe`. Den skal arve `AbstraktListe` og kan lages helt maken til klassen `TabellBeholder` (se *Avsnitt* 3.1.3). Metoden `leggInn(int indeks, T t)` må kodes her istedenfor metoden `leggInn(T t)`. Hvis tabellen er full, skal den «utvides» (bruk metoden `copyOf` fra klassen `Arrays`). Også den indre iteratorklassen lages helt maken til den i `TabellBeholder`, men den burde her hete `EnkelTabellListeIterator`.

3.2.2 En tabellbasert liste

Vi skal implementere grensesnittet `Liste` ved hjelp av en tabell. Vi trenger to variabler – en tabell og et antall. Vi kaller klassen `TabellListe`:

```
public class TabellListe<T> implements Liste<T>
{
  private T[] a;
  private int antall;

  // konstruktører og metoder kommer her
}
```

Programkode 3.2.2 a)

Konstruktører Vi trenger en som gir oss muligheten til å oppgi størrelsen på den interne tabellen og en standardkonstruktør som oppretter en tabell med fast startstørrelse. Videre er det gunstig ha ha en konstruktør som lager en `TabellListe` ved hjelp av verdiene i en tabell:

```
@SuppressWarnings("unchecked")          // pga. konverteringen: Object[] -> T[]
public TabellListe(int størrelse)       // konstruktør
{
  a = (T[])new Object[størrelse];       // oppretter tabellen
  antall = 0;                           // foreløpig ingen verdier
}

public TabellListe()                    // standardkonstruktør
{
  this(10);                             // startstørrelse på 10
}
```

Programkode 3.2.2 b)

Når vi skal teste er det greit å kunne sette opp en tabell og så bruke dens verdier som testverdier. Derfor burde vi ha en konstruktør med en tabell som parameter. Dvs. slik:

```
String[] navn = {"Per", "Kari", "Ole", "Azra"};
Liste<String> liste = new TabellListe<>(navn);
```

Programkode 3.2.2 c)

Et problem som da dukker opp, er at tabellen kan inneholde null-verdier og det skal vi ikke ha i vår `TabellListe`. Da kan vi enten forkaste hele tabellen eller vi kan lage en `TabellListe` av de verdiene fra tabellen som ikke er null. Her velger de siste:

```
public TabellListe(T[] b)                   // en T-tabell som parameter
{
  this(b.length);                           // kaller den andre konstruktøren

  for (T verdi : b)
  {
    if (verdi != null) a[antall++] = verdi;  // hopper over null-verdier
  }
}
```

Programkode 3.2.2 d)

En **aksessor** er en metode som henter «informasjon» fra en instans av en klasse uten å endre på dens tilstand (endrer ingen instansvariabler). `Liste` har de fem aksessorene *inneholder*(), *hent*(), *indeksTil*(), *antall*() og *tom*() og i tillegg *toString*().

```
public int antall()
{
  return antall;          // returnerer antallet
}

public boolean tom()
{
  return antall == 0;     // listen er tom hvis antall er 0
}
```

Programkode 3.2.2 e)

Metoden `hent(int indeks)` har en indeks som parameterverdi. Grensesnittet `Liste` har en metode som sjekker den for oss og kaster en *IndexOutOfBoundsException* hvis indeksen er ulovlig. Koden for metoden `hent` blir dermed svært enkel:

```
public T hent(int indeks)
{
  indeksKontroll(indeks, false);   // false: indeks = antall er ulovlig
  return a[indeks];                // returnerer er tabellelement
}
```

Programkode 3.2.2 f)

Metoden *indeksTil*(`T verdi`) skal returnere indeksen til første forekomst av `verdi` og -1 hvis den ikke er der. Metoden *inneholder*(`T verdi`) kan kodes ved hjelp av *indeksTil*():

```
public int indeksTil(T verdi)
{
  for (int i = 0; i < antall; i++)
  {
    if (a[i].equals(verdi)) return i;   // funnet!
  }
  return -1;   // ikke funnet!
}

public boolean inneholder(T verdi)
{
  return indeksTil(verdi) != -1;
}
```

Programkode 3.2.2 g)

Metoden *toString*() er ikke satt opp i grensesnittet `Liste`, men den arver vi uansett fra basisklassen `Object`. Den må alltid overskrives. Se *Oppgave* 3 under.

Oppgaver til Avsnitt 3.2.2

1. Legg inn klassen `public class TabellListe<T> implements Liste<T>` under package `hjelpeklasser` og legg inn de metodene som er laget i dette avsnittet.
2. Lag en String-tabell *s*, en instans av `TabellListe` med *s* som parameter og sjekk at aksessor-metodene virker som de skal. Sjekk at det virker hvis *s* har en eller flere null-er.
3. Lag metoden `public String toString()`. Hvis listen f.eks. inneholder 1, 2 og 3, skal tegnstrengen bli lik "[1, 2, 3]". Tom liste skal gi "[]". Bruk en `StringBuilder` eller en `StringJoiner`. Bruk metoden til å skrive ut skrive ut de tabellistene du laget i Oppgave 2.
4. La klassen få konstruktøren `public TabellListe(Iterable<T> itererbar)`. Bruk først `this();` Hent så én og én verdi ved hjelp av iteratoren i `itererbar` og legg dem fortløpende inn i tabellen. Øk med 50% hvis den blir full.

3.2.3 Mutatorer

En metode som kan endre tilstanden (dvs. endre på noen av instansvariablene) til en instans av en klasse, kalles en *mutator*. Det kommer av verbet *mutate* som betyr å endre. Vi bruker det også på norsk, f.eks. i ordene *mutere* og *mutasjon*. I grensesnittet Liste har vi de seks mutatorene `leggInn` (to stykker), `oppdater`, `fjern` (to stykker) og `nullstill`.

I denne klassen tillates duplikater, men ikke null-verdier. Vi kan sjekke parameterverdien ved hjelp av metoden `requireNonNull()` i klassen `Objects`.

Den første `leggInn`-metoden skal legge verdien bakerst i listen. Den andre har i tillegg en indeks som parameter. Det betyr at etter innleggingen skal den nye verdien ha den posisjonen eller indeksen i listen som parameterverdien sier. Det betyr at alle verdiene som kommer etter der vi legger inn verdien, får en posisjon som er én mer enn sist. Her bruker vi *true* som argument i *`indeksKontroll`*`()` for å sjekke om indeksen er lovlig siden en indeks lik `antall` nå er tillatt. Det betyr at verdien da skal legges bak de som allerede er der:

```
public boolean leggInn(T verdi)  // inn bakerst
{
  Objects.requireNonNull(verdi, "null er ulovlig!");

  if (antall == a.length)  // En full tabell utvides med 50%
  {
    a = Arrays.copyOf(a,(3*antall)/2 + 1);
  }

  a[antall++] = verdi;    // setter inn ny verdi
  return true;            // vellykket innlegging
}

public void leggInn(int indeks, T verdi)
{
  Objects.requireNonNull(verdi, "null er ulovlig!");
  indeksKontroll(indeks, true);  // true: indeks = antall er lovlig

  // En full tabell utvides med 50%
  if (antall == a.length) a = Arrays.copyOf(a,(3*antall)/2 + 1);

  // rydder plass til den nye verdien
  System.arraycopy(a, indeks, a, indeks + 1, antall - indeks);

  a[indeks] = verdi;     // setter inn ny verdi
  antall++;              // vellykket innlegging
}
```

Programkode* 3.2.3 *b)

Kodingen for metodene *oppdater* og *fjern* blir nærmest rett frem:

```
public T oppdater(int indeks, T verdi)
{
  Objects.requireNonNull(verdi, "null er ulovlig!");
  indeksKontroll(indeks, false);  // false: indeks = antall er ulovlig

  T gammelverdi = a[indeks];      // tar vare på den gamle verdien
  a[indeks] = verdi;              // oppdaterer
  return gammelverdi;             // returnerer den gamle verdien
}
```

```
public T fjern(int indeks)
{
  indeksKontroll(indeks, false);  // false: indeks = antall er ulovlig
  T verdi = a[indeks];

  antall--; // sletter ved å flytte verdier mot venstre
  System.arraycopy(a, indeks + 1, a, indeks, antall - indeks);
  a[antall] = null;   // tilrettelegger for "søppeltømming"

  return verdi;
}
```

Programkode 3.2.3 c)

De eneste metodene, bortsett fra iteratoren, som ennå ikke har fått kode, er metodene *public boolean fjern(T verdi)* og *public void nullstill()*. Den første skal fjerne *verdi* fra listen og den andre skal «tømme» listen. Se *Oppgave 1 og 2*.

Oppgaver til Avsnitt 3.2.3

1. Lag metoden *public boolean fjern(T verdi)*. Se grensesnittet `Liste`. Metoden skal fjerne **første** forekomst av `verdi` og returnere *true* hvis fjerningen var vellykket og returnere *false* hvis `verdi` ikke finnes i listen.
2. Lag metoden *public void nullstill()*. Se gresnesnittet `Liste`. Den skal «tømme» listen, dvs. «tømme» den interne tabellen og samtidig sørge for at de elementene som måtte ligge igjen går til «resirkulering». Hvis den interne tabellen har en lengde som er større enn 10, skal den erstattes med en som har lengde 10.

3.2.4 En indre iteratorklasse

Den siste metoden i grensesnittet `Liste` heter `iterator()` og skal returnere en `Iterator<T>`. Vi løser dette ved å lage en indre klasse som implementere grensesnittet `Iterator`. Se *Avsnitt* 3.1.1. Det at den er en indre klasse betyr at den er satt opp inne i klassen `TabellListe<T>`. En viktig fordel med en indre klasse er at den ytre klassens (her class TabellListe<T>) variabler er «synlige» fra den indre klassens metoder.

Metoden `next()` i iteratoren skal returnere en ny verdi for hvert kall og kravet er at verdiene skal komme i samme rekkefølge som verdiene har med hensyn på listeindekseringen. Her vil det si fra venstre mot høyre i den interne tabellen *a*. I figuren under har tabellen plass til 16 verdier (lengden er 16) og det er lagt inn 12 stykker. Det betyr at variabelen *antall* er lik 12.

10	7	14	2	10	21	8	3	7	13	9	6				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Figur 3.2.4 a) : En tabell med 12 verdier og med plass til 16 verdier

Iterator-klassen må ha en instansvariabel denne som holder rede på hvor aktuell verdi er, dvs. den verdien som et kall på `next()` skal returnere. Vi må også passe på at denne ikke er utenfor den «lovlige» delen av tabellen, dvs. at vi må ha `denne < antall`. I *Figur* 3.2.4 *a)* over betyr det at vi må ha `denne < 12`. Det gjøres i metoden `hasNext()`. Alt dette legges i iterator-klassen `TabellListeIterator`:

```
// Skal ligge som en indre klasse i class TabellListe
private class TabellListeIterator implements Iterator<T>
{
  private int denne = 0;       // instansvariabel

  public boolean hasNext()     // sjekker om det er flere igjen
  {
    return denne < antall;     // sjekker verdien til denne
  }

  public T next()              // returnerer aktuell verdi
  {
    if (!hasNext())
      throw new NoSuchElementException("Tomt eller ingen verdier igjen!");
    return a[denne++]; // a[denne] returneres før denne++
  }
}  // TabellListeIterator
```

Programkode* 3.2.4 *a)

Klassen `TabellListeIterator` skal som nevnt være en indre klasse i `TabellListe`. Den kan f.eks. legges rett foran metoden `iterator()` og den metoden kodes slik:

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

Programkode* 3.2.4 *b)

Metoden *remove*() i klassen `TabellListeIterator` står igjen. Den skal fjerne verdien som ble returnert av det siste kallet på *next*(). Den kan ikke kalles to ganger på rad (se *Tabell* 3.1.1).

La *fjernOK* være ny instansvariabel i iteratoren. Den settes til *true* etter et kall på *next*() og til false etter et kall på *remove*(). Metoden *next*() må endres:

```
private boolean fjernOK = false;   // ny instansvariabel i TabellListeIterator

public T next()                    // ny versjon
{
  if (!hasNext())
    throw new NoSuchElementException("Tomt eller ingen verdier igjen!");

  T denneVerdi = a[denne];   // henter aktuell verdi
  denne++;                   // flytter indeksen
  fjernOK = true;            // nå kan remove() kalles

  return denneVerdi;         // returnerer verdien
}

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

  fjernOK = false;           // remove() kan ikke kalles på nytt

  // verdien i denne - 1 skal fjernes da den ble returnert i siste kall
  // på next(), verdiene fra og med denne flyttes derfor en mot venstre
  antall--;           // en verdi vil bli fjernet
  denne--;            // denne må flyttes til venstre

  System.arraycopy(a, denne + 1, a, denne, antall - denne);  // tetter igjen
  a[antall] = null;   // verdien som lå lengst til høyre nulles
}
```

Programkode* 3.2.4 *c)

Eksempel: Ta utgangspunkt i *Tabell* 3.2.4 *a)* med denne = 6 (tabellverdi 8). Dermed er det tallet 21 (indeks 5) som skal fjernes. Det gjøres ved at alle verdiene til høyre for indeks 5 flyttes mot venstre. Siste verdi (indeks antall - 1 = 11) blir liggende igjen og må «nulles». Deretter må både denne og antall oppdateres. Dermed får vi dette resultatet:

10	7	14	2	10	8	3	7	13	9	6					
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Figur 3.2.4 b) : Verdien som lå i posisjon 5 er fjernet og denne er nå lik 5

Oppgaver til Avsnitt 3.2.4

1. Legg iterator-klassen og metoden iterator() fra *Avsnitt* 3.2.4 inn i TabellListe. Pass på at det er de siste versjonene av next() og remove() (*Programkode* 3.2.4 *c*) du bruker.
2. Test metodene i TabellListe og de arvede metodene *fjernHvis*() og *forEach*().
3. Metoden *fjernHvis*() arves fra Beholder. Kod den i TabellListe og da direkte uten å bruke iteratorens remove-metode.
4. Klassen TabellListe arver metoden *forEach*() fra Iterable. Kod den i TabellListe og da direkte uten å bruke iteratoren.
5. Kod metoden *forEachRemaining*() direkte i TabellListeIterator.

3.2.5 Flere iteratorer samtidig

Hvis en liste endres etter at en iterator har startet, kan vi få uventede resultater:

```
String[] s = {"Per","Kari","Ole"};

Liste<String> liste = new TabellListe<>();

for (String navn : s) liste.leggInn(navn);

System.out.println(liste);

// Utskrift: [Per, Kari, Ole]
```

Programkode 3.2.5 a)

I *Programkode* 3.2.5 *a)* får vi som ventet at listen inneholder Per, Kari og Ole. Men hva skjer når vi legger til flg. setninger:

```
Iterator<String> i = liste.iterator();     // oppretter en iterator
System.out.println(i.next());              // den første i listen

liste.fjern("Per");                        // fjerner den første
System.out.println(i.next());              // den neste i listen
```

Programkode 3.2.5 b)

I første setning i *Programkode* 3.2.5 *b)* «startes» en iterator og kallet `next()` gir oss den første i listen, dvs. Per. Deretter fjernes Per. Neste kall på `next()` burde logisk sett gi oss Kari, men det er isteden Ole som blir skrevet ut. Det kommer av at fjern-metoden har flyttet Ole til den plassen som Kari hadde. Men det kan ikke iteratoren vite. Dette viser at vi kan få uventede resultater hvis det gjøres endringer i en liste etter at en iterator er opprettet.

En listeendring kan skje ved et kall på en av mutatorene (dvs. `leggInn`, `fjern`, `oppdater`, `nullstill`), men også ved å bruke `remove` fra en annen iterator. Det er fullt mulig å ha flere iteratorer på samme liste, men da vil et kall på `remove` i en av dem få konsekvenser for de andre. Hva skjer hvis flg. setninger legges til i *Programkode* 3.2.5 *a)*?

```
Iterator<String> i = liste.iterator();     // oppretter en iterator i
Iterator<String> j = liste.iterator();     // oppretter en iterator j

System.out.println(i.next());              // den første i listen
i.remove();                                // fjerner den første
System.out.println(j.next());              // den første i listen
```

Programkode 3.2.5 c)

I *Programkode* 3.2.5 *c)* fjernes den første (Per) ved hjelp av `remove` i iteratoren *i*. Men det første kallet på `next` i iteratoren *j* burde logisk sett gi oss Per, men isteden får vi Kari.

En mulig måte å komme ut av dette på er å si at når en iterator først er satt i gang, er det ikke «tillatt» å gjøre endringer, hverken av en av listens mutatorer eller av `remove` i en annen iterator. Det kan f.eks. løses ved at det i så fall kastes unntak. Det er imidlertid rimelig å ha ett tilfelle der «forbudet» ikke gjelder. Hvis vi har kun én iterator, bør det være tillatt å kalle iteratorens `remove` siden den fjerner en verdi som vi allerede har sett eller fått ved hjelp av *`next`*`()`. Dermed vil den videre traverseringen ikke gi noen uventede resultater.

Dette løses ved to nye instansvariabler: *endringer* i TabellListe og *iteratorendringer* i TabellListeIterator. Der skal *endringer* start med 0, mens *iteratorendringer* skal ha verdien til *endringer* som startverdi. Starten på de to klassene vil derfor bli slik:

```
public class TabellListe<T> implements Liste<T>
{
  private T[] a;
  private int antall;
  private int endringer;    // ny variabel

  // øvrige ting
}

private class TabellListeIterator implements Iterator<T>
{
  private int denne = 0;
  private boolean fjernOK = false;
  private int iteratorendringer = endringer;  // ny variabel

  // øvrige ting
}
```

Programkode* 3.2.5 *d)

For at dette skal virke etter hensikten må *endringer* økes med 1 i alle mutatorene. Se *Oppgave* 2. Metoden *next*() i TabellListeIterator sjekker om de to endringsvariablene er like. Hvis ikke kastes en ConcurrentModificationException. Det samme gjøres i metoden remove, men her skal begge økes med 1. Dermed vil de fortsatt være like etter et kall på remove. Det gir flg. utvidelse av *next*() og *remove*() i *Programkode* 3.2.4 *c)*:

```
public T next()
{
  if (iteratorendringer != endringer)
  {
    throw new ConcurrentModificationException("Listen er endret!");
  }

  if (!hasNext())
  {
    throw new NoSuchElementException("Tomt eller ingen verdier igjen!");
  }

  T denneVerdi = a[denne];   // henter aktuell verdi
  denne++;                   // flytter indeksen
  fjernOK = true;            // nå kan remove() kalles
  return denneVerdi;         // returnerer verdien
}

public void remove()
{
  if (iteratorendringer != endringer) throw new
    ConcurrentModificationException("Listen er endret!");

  if (!fjernOK) throw
    new IllegalStateException("Ulovlig tilstand!");

  fjernOK = false;           // remove() kan ikke kalles på nytt
```

```
    // verdien i denne - 1 skal fjernes da den ble returnert i siste kall
    // på next(), verdiene fra og med denne flyttes derfor en mot venstre

    antall--;           // en verdi vil bli fjernet
    denne--;            // denne må flyttes til venstre

    // tetter igjen
    System.arraycopy(a, denne + 1, a, denne, antall - denne);

    a[antall] = null;   // verdien som lå lengst til høyre nulles

    endringer++;
    iteratorendringer++;
  }
```

Programkode* 3.2.5 *e)

Oppgaver til Avsnitt 3.2.5

1. Legg instansvariabelen *endringer* i klassen `TabellListe` og instansvariabelen *iteratorendringer* (med startverdi) i den indre klassen `TabellListeIterator`. Se *Programkode* 3.2.5 *d)*. Legg så inn de nye versjonene av *next*() og *remove*() i `TabellListeIterator`. Se *Programkode* 3.2.5 *e)*.
2. Sørg for at variabelen *endringer* økes i alle mutatorene, dvs. i *leggInn*-metodene, i *fjern*-metodene, i metodene *oppdater*() og *nullstill*().
3. Sjekk hva resutatet nå blir for *Programkode* 3.2.5 *a)*, *b)* og *c)*.

3.2.6 Klassen ArrayList i java.util

Klassen `ArrayList` er deklarert slik:

```
public class ArrayList<T> extends AbstractList<T>
       implements List<T>, RandomAccess, Cloneable, java.io.Serializable
```

Dermed har den alle metodene som er satt opp i grensesnittet `List` i *Programkode* 3.2.1 *b)* og enda flere metoder siden grensesnittet `List` har flere metoder enn det som står der. Se `java.util.List`. Her er noen av de metodene som mangler i *Programkode* 3.2.1 *b)*:

- `boolean addAll(Collection<? extends T> c)`
- `boolean addAll(int index, Collection<? extends T> c)`
- `boolean containsAll(Collection<?> c)`
- `int lastIndexOf(Object o)`
- `ListIterator<T> listIterator()`
- `ListIterator<T> listIterator(int index)`
- `boolean removeAll(Collection<?> c)`
- `<T> T[] toArray(T[] a)`
- `Object[] toArray()`

Her er eksempler på hvordan noen av metodene kan brukes:

```
public static void main(String[] args)
{
  List<Integer> liste1 = new ArrayList<>();
  for (int i = 1; i <= 5; i++) liste1.add(i);  // tallene fra 1 til 5

  List<Integer> liste2 = new ArrayList<>();
  for (int i = 6; i <= 10; i++) liste2.add(i);  // tallene fra 6 til 10

  liste1.addAll(0,liste2);   // liste2 legges forrest i liste 1
  System.out.println(liste1);  // [6, 7, 8, 9, 10, 1, 2, 3, 4, 5]

  liste1.removeAll(liste2);   // fjerner tallene fra 6 til 10
  System.out.println(liste1);  // [1, 2, 3, 4, 5]

  ListIterator<Integer> i = liste1.listIterator(4);  // bakerst
  while (i.hasPrevious()) System.out.print(i.previous() + " ");
  System.out.println();  // 4 3 2 1

  Integer[] a = new Integer[liste1.size()];
  Integer[] b = liste1.toArray(a);
  System.out.println(Arrays.toString(a));  // [1, 2, 3, 4, 5]
}
```

Programkode* 3.2.6 *a)

En vanlig iterator har metodene `hasNext`, `next` og `remove`. `ArrayList` har en vanlig iterator, men også en spesiell listeiterator. Den kan gå begge veier - fremover ved hjelp av `hasNext` og `next` og bakover ved `hasPrevious` og `previous`. Den kan også starte hvor som helst i listen ved å oppgi en indeks.

Metoden `<T> T[] toArray(T[] a)` legger listens innhold i tabellen *a* hvis det er plass. Hvis ikke, returneres en tabell med innholdet. Tabellen *a* og returtabellen er identiske hvis det er plass i *a*.

3.2.7 En samleklasse for liste- og beholdermetoder

Klassen `Tabell` som ble satt opp første gang i *Avsnitt* 1.2.2 , var/er tenkt å være en felles klasse for ulike tabellmetoder. Det svarer til klassen *Arrays* i *java.util*. Java har også en samleklasse for `list-` og `collection`-metoder. Den heter *Collections*. Hvis vi lager en slik klasse selv, vil det bli enklere å forstå oppbyggingen av *Collections*. Den skal inneholde metoder som kan brukes på objekter som er `Iterable`, er en `Beholder` eller er en `Liste`. Vi kaller den `Beholdere` og legger den under katalogen/mappen (package) `hjelpeklasser`:

```
package hjelpeklasser;

import java.util.*;

public class Beholdere
{
  private Beholdere() { }  // hindrer instansiering

  public static <T> T maks(Iterable<T> itererbar, Comparator<? super T> c)
  {
    Iterator<T> it = itererbar.iterator();  // henter iteratoren

    if (!it.hasNext())
      throw new NoSuchElementException("Ingen verdier!");

    T maksverdi = it.next();  // finnes siden listen ikke er tom

    while (it.hasNext())
    {
      T verdi = it.next();
      if (c.compare(verdi,maksverdi) > 0) maksverdi = verdi;
    }
    return maksverdi;
  }
} // class Beholdere
```

Programkode* 3.2.7 *a)

Oppgaver til Avsnitt 3.2.7

1. Flytt *Programkode* 3.2.7 *a)* over til deg og legg den under package `hjelpeklasser`.
2. Lag et program der du først oppretter en `TabellListe` og så legger inn en samling verdier i listen. Bruk så metoden `maks` fra klassen `Beholdere` til å finne den største verdien i listen. Hvis datatypen din er `Comparable`, kan du bruke *naturalOrder*() fra `Comparator`.
3. Lag metoden `public static int frekvens(Iterable<?> itererbar, Object o)`. Den skal returnere antallet forekomster av objektet o. Bruk metoden `equals`. Legg den i samleklassen `Beholdere`. Test metoden på en instans av klassen `TabellListe` der du har flere like verdier. Dette vil virke siden en `Liste` er `Iterable`.
4. Gjør om metoden `maks` fra *Programkode* 3.2.7 *a)* slik at den får signaturen: `public static <T> int maks(Liste<T> liste, Comparator<? super T> c)`. Den skal nå returnere posisjonen til den største verdien i listen. Bruk fortsatt iteratoren i letingen. Legg den i samleklassen `Beholdere`. Test metoden på en instans av klassen `TabellListe`.