Generiske beholdere

IN1010 – Objektorientert programmering

Lars Tveito

Institutt for informatikk, Universitetet i Oslo
in1010@ifi.uio.no

Introduksjon

Plan for forelesningen

- Vi avslutter fortellingen om lenkede lister
  - Vi skal se på *dobbeltlenkede* lister
  - *Kort* om hode- og halenoder
  - En liten oversikt som viser når ulike implementasjoner egner seg
- Vi gir en introduksjon til *generiske* klasser i Java
  - Beholdere er den vanligste bruksområdet for generiske klasser
- Vi ser på iteratorer, som er klasser som implementerer grensesnittet `Iterable`
- Vi ser litt på unntak, og hvordan vi kan lage våre egne

Dobbeltlenket liste

Dobbeltlenket liste

- Ved å knytte noder sammen med *neste*- og *forrige*-pekere får vi en *dobbeltlenket liste*
- Vi kan lage lister som objekter som har pekere til den første og den siste noden
- En dobbeltlenket liste gjør både innsetting og sletting på begge ender av listen raskt og forholdsvis enkelt

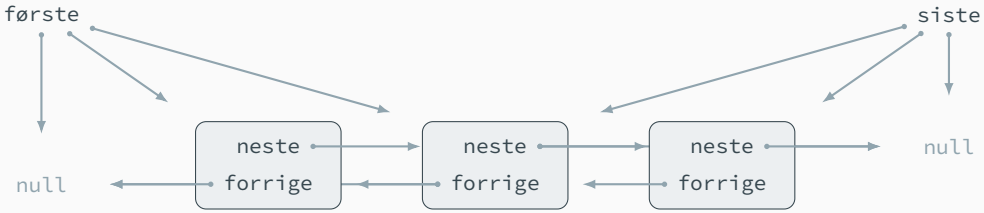
Klassedeklarasjon

- I en dobbeltlenket liste bruker vi en *indre* nodeklasse
- Hver node har en peker til både forrige og neste node (samt elementet)
- Vi lar listen både ha peker til *første* og *siste* node
- Vi holder også styr på antall

Klassedeklarasjon (kode)

```
1  class DobbeltLenketStrengListe implements StrengListe {
2
3      private class Node {
4          String elem;
5          Node forrige, neste;
6
7          Node(Node forrige, String elem, Node neste) {
8              this.forrige = forrige;
9              this.elem = elem;
10             this.neste = neste;
11         }
12     }
13
14     private Node første;
15     private Node siste;
16     private int antall;
17 }
```

Illustrasjon



Sett inn først

- Når vi setter inn først i en dobbeltlenket liste må vi:
  - Lage en ny node
  - La den nye noden sin forrige peke på `null`
  - La den nye noden sin neste peke på det `første` peker på
  - Hvis listen er tom må `siste` peke på den nye noden
  - Hvis listen ikke er tom må `første` sin forrige peke på den nye noden
  - Til slutt settes `første` til å peke på den nye noden og antallet økes

Sett inn først (kode)

```java
@Override
public void settInnFørst(String elem) {
    Node ny = new Node(null, elem, første);
    if (første == null) {
        siste = ny;
    } else {
        første.forrige = ny;
    }
    første = ny;
    antall++;
}
```

Sett inn sist

- Vi ønsker å sette inn *sist* i en dobbeltlenket liste
- Merk at dette blir helt symmetrisk med å sette inn først
  - Fordi vi både har første og siste, og forrige og neste

```
1  @Override
2  public void settInnSist(String elem) {
3      Node ny = new Node(siste, elem, null);
4      if (siste == null) {
5          første = ny;
6      } else {
7          siste.neste = ny;
8      }
9      siste = ny;
10     antall++;
11 }
```

Fjern første

- Vi setter `første` til å peke på den sin neste (med mindre listen er tom)
- Hvis den siste noden ble slettet, settes `siste` også til `null`
- Ellers oppdateres `første` sin forrige til å peke på `null`

```java
@Override
public void fjernFørste() {
    if (første != null) {
        første = første.neste;
        antall--;
    }
    if (første == null) {
        siste = null;
    } else {
        første.forrige = null;
    }
}
```

Fjern siste

- Å fjerne siste node er igjen helt symmetrisk med å fjerne første

```
@Override
public void fjernSiste() {
    if (siste != null) {
        siste = siste.forrige;
        antall--;
    }
    if (siste == null) {
        første = null;
    } else {
        siste.neste = null;
    }
}
```

Hente på indeks

- For å hente på indeks i en dobbeltlenket liste kan vi velge om å følge forrige-pekere eller neste-pekere
- Hvis `indeks` er *under* halvparten av `antall` bør vi følge *neste*-pekere fra *første*
- Hvis `indeks` er *over* halvparten av `antall` bør vi følge *forrige*-pekere fra *siste*

Hode- og halenoder

Hode- og halenoder

- Til nå har vi sett på lister med peker til første, og (eventuelt) siste node
- De fleste metoder må håndtere tomme lister spesielt
  - Ved innsetting *først* i en tom liste, må også `siste` oppdateres
  - Ved fjerning *først* i en liste med ett element, må også `siste` oppdateres
- Mye av logikken kan forenkles *litt* ved å heller ha to spesielle noder:
  - Nodene kalles henholdsvis *hode* og *hale*
  - De holder ikke på noe element
  - De kan ikke slettes
  - De er alltid på hver sin ende av listen
- Merk at en liste med eller uten hode og hale vil oppføre seg omtrent likt
- Vi refererer til `hode.neste` som den første noden og `hale.forrige` som den siste noden

Klassedefinisjon

```
1  class DobbeltLenketStrengListeHodeHale implements StrengListe {
2
3      private class Node { /* ··· */ }
4
5      private Node hode;
6      private Node hale;
7      private int antall;
8
9      DobbeltLenketStrengListeHodeHale() {
10         hode = new Node(null, null, null);
11         hale = new Node(hode, null, null);
12         hode.neste = hale;
13     }
14 }
```

`hode` og `hale` opprettes sammen med listen, og peker på hverandre

Sett inn først

- Når vi nå skal sette inn først vil vi sette inn elementet *mellom* hodet og den første noden
- Vi trenger ikke et spesialtilfelle for når listen er tom

```
1 @Override
2 public void settInnFørst(String elem) {
3     Node ny = new Node(hode, elem, hode.neste);
4     hode.neste.forrige = ny;
5     hode.neste = ny;
6     antall++;
7 }
```

Fjern først

- Når vi nå skal fjerne første vil vi fjerne hode sin neste
- Her må vi fremdeles passe på at vi ikke sletter fra en tom liste
- En liste er tom hvis `hode.neste` peker på halen til listen

```java
@Override
public void fjernFørste() {
    Node første = hode.neste;
    if (første != hale) {
        første.neste.forrige = hode;
        hode.neste = første.neste;
        antall--;
    }
}
```

Oppsummering av listeimplementasjoner

Implementasjonene vi har sett

Vi har sett en deler av mange ulike listeimplementasjoner

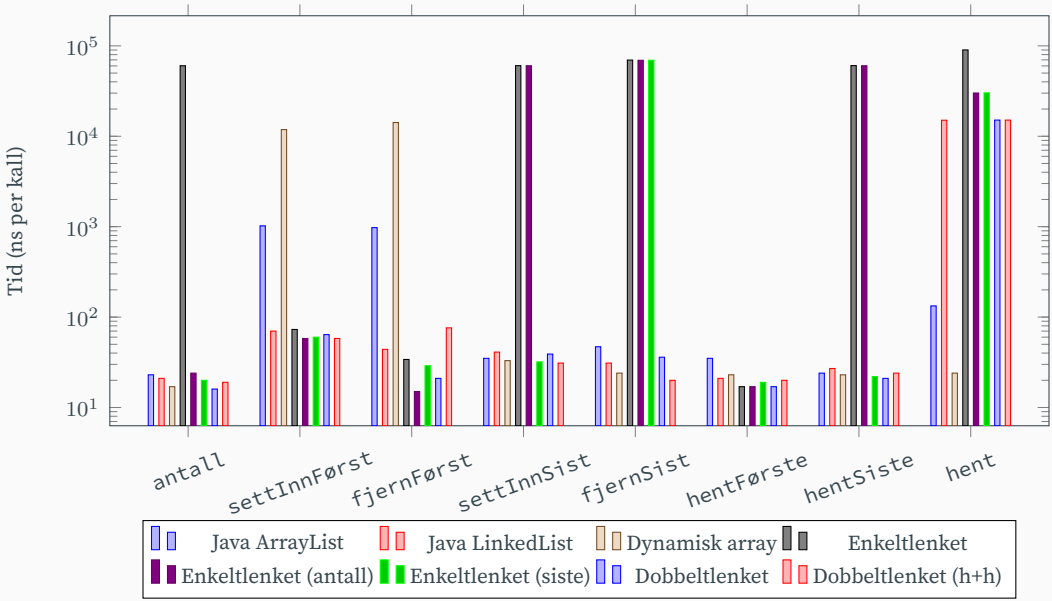
Fil	Linjer
`DobbeltLenketStrengListe.java`	98
`EnkeltLenketStrengListeMedSiste.java`	92
`EnkeltLenketStrengListe.java`	90
`DobbeltLenketStrengListeHodeHale.java`	89
`EnkeltLenketStrengListeMedAntall.java`	89
`DynamiskStrengArray.java`	87

Tidtakning (for de spesielt interesserte)

- Under ser vi antall nanosekunder hver operasjon bruker *i snitt*
  - Hver operasjon blir kalt på ti tusen ganger
  - Operasjonene som *ikke endrer* på listen blir kalt på når listen inneholder fem tusen elementer
- Ta tallene med en klype salt
- Vises som et diagram på neste slide

Implementasjon	`antall`	`settInnFørst`	`fjernFørst`	`settInnSist`	`fjernSist`	`hentFørste`	`hentSiste`	`hent`
Java ArrayList	23	1022	976	35	47	35	24	133
Java LinkedList	21	70	44	41	31	21	27	15051
Dynamisk Array	17	11857	14191	33	24	23	23	24
Enkeltlenket	60290	73	34	60398	69650	17	60502	90161
Enkeltlenket (antall)	24	58	15	60250	69266	17	60266	30093
Enkeltlenket (siste)	20	60	29	32	69381	19	22	30222
Dobbeltlenket	16	64	21	39	36	17	21	15104
Dobbeltlenket (h+h)	19	58	76	31	20	20	24	15098

Tidtakning (diagram)



Hvilke type lister egner seg til hva?

- Alle listene vi har sett på egner seg som en *stack*
  - Skal man implementere en bør man antageligvis velge det *enkleste*
  - En enkeltlenket liste med kun peker til første node er antageligvis det enkleste
- En enkeltlenket liste trenger en sistepeker for at den skal kunne egne seg for å være en *kø*
- Java sin `ArrayList` er veldig rask på det meste, og er ofte det du vil bruke
  - Definitivt det du vil bruke dersom du vil så opp på indeks ofte
  - Merk at dynamiske arrayer er raske i *snitt*, men enkelte innsettinger vil gå tregt
- Her vil kun en dobbeltlenket liste egne seg til å kunne brukes som en *deque*
  - En *deque* støtter innsetting og sletting på begge sider av listen
  - Det blir tema for neste fellesøvelse!

Generiske typer

En mer generell `StrengListe`

- Så langt har vi sett på ulike varianter av lister over strenger
- Her er et utsnitt av grensesnittet hvor `String` forekommer som parameter eller returverdi

```
1  interface StrengListe {
2      public void settInnFørst(String elem);
3      public String hentFørste();
4      // ···
5  }
```

- Husk at alle objekter i Java arver fra `Object`
- Kanskje vi kunne laget en liste som tar og returnerer `Object` i stedet?

Lister av objekter

- Et grensesnitt for lister av objekter kan se slik ut

```
interface Liste {
    public void settInnFørst(Object elem);
    public Object hentFørste();
    // ...
}
```

- Anta at vi har en klasse `EnkeltLenketListe` som implementerer grensesnittet
- Den kan instansieres slik:
  `Liste liste = new EnkeltLenketListe();`
- Vi kan sette inn elementer slik:
  `liste.settInnFørst("Dette er en streng");`

Lister av objekter medfører kjøretidsfeil

- Skal vi hente elementer ut må vi bruke typekonvertering:
  `String s = (String) liste.hentFørste();`
- Typekonvertering kan feile *under kjøring*
  - Slike feil er vanskeligere å håndtere, spesielt når programmet blir større
- `Person p = (Person) liste.hentFørste();`
  - Dette vil kræsje
- I 2004 ble Java utvidet med *generiske typer* som en løsning på dette problemet

Generiske klasser og grensesnitt

- I Java kan vi definere nye typer gjennom klasser og grensesnitt
- En klasse eller et grensesnitt kan ta *typeparametere* (også kalt *typevariabler*)
  `class EnkeltLenketListe<T> { /* ··· */ }`
  - Her er `T` en typeparameter til klassen `EnkeltLenketListe`
- En klasse eller et grensesnitt kan ta flere typeparametere
- `interface Ordbok<K, V> { /* ··· */ }`
  - Her er `K` og `V` typeparametere til et hypotetisk grensesnitt `Ordbok`
- Vi sier klassen eller grensesnittet er generisk, hvis den tar én eller flere typeparametere
- Java anbefaler at typevariabler består av én stor bokstav

Generisk grensesnitt `Liste<T>`

- Vi definerer grensesnittet `Liste<T>` der `T` forekommer som *typeparameter*
- I definisjonen av grensesnittet kan vi nå bruke `T` som en *typevariabel*
- Der vi før ville skrevet `String`, skriver vi heller `T`

```
1  interface Liste<T> {
2      public int antall();
3      public void settInnFørst(T elem);
4      public void settInnSist(T elem);
5      public void fjernFørste();
6      public void fjernSiste();
7      public T hentFørste();
8      public T hentSiste();
9      public T hent(int indeks);
10 }
```

Instansiering av generiske klasser

- En generisk klasse eller grensesnitt må instansieres med et *typeargument*
  ```
  Liste<String> liste = new EnkeltLenketListe<>();
  ```
  - Her forekommer `String` som et typeargument til `Liste`
  - Typeargumentet blir implisitt gitt til `EnkeltLenketListe` ved `<>`
- Vi trenger ikke lenger bruke typekonvertering:
  ```
  String s = liste.hentFørste();
  ```
- Og typefeil vil gi *kompileringsfeil*
  ```
  Person p = liste.hentFørste();
  ```

Lite utdrag fra `EnkeltLenketListe<T>`

```
class EnkeltLenketListe<T> implements Liste<T> {

    private class Node {
        T elem;
        Node neste;
        Node(T elem, Node neste) {
            this.elem = elem;
            this.neste = neste;
        }
    }

    @Override
    public void settInnFørst(T elem) {
        Node ny = new Node(elem, første);
        første = ny;
    }
}
```

Generiske typer og arv

- Noen ganger ønsker vi å *begrense* en generisk type
- Da kan vi bruke nøkkelordet `extends` i typedeklarasjonen
- For eksempel, gitt en slik generisk klassedeklarasjon:
  `class TallBoks<N extends Number> { /* ··· */ }`
- Klassen `TallBoks` må instansieres med et typeargument som arver fra `Number`
- Dette kompilerer:
  `TallBoks<Double> tallboks = new TallBoks<>();`
- Dette kompilerer ikke:
  `TallBoks<String> dumboks = new TallBoks<>();`
- Et mye brukt eksempel er å begrense en klasse til å ta en *sammenlignbar* type
  `class SortertListe<T extends Comparable<T>> { /* ··· */ }`
  - `T` må være en type som er sammenlignbar med elementer av typen `T`

Iteratorer

Itererbare klasser

- Et objekt av en klasse som implementerer grensesnittet `Iterable<T>` er *itererbar*
- Å iterere over et objekt vil si å *løpe gjennom elementene*
- En beholder er også som regel itererbar
- Hvis `strengliste` er en itererbar liste, så kan vi gå gjennom strengene slik:

```
1 for (String s : strengliste) {
2     System.out.println(s);
3 }
```

- Grensesnittet `Iterable<T>` krever kun implementasjon av én metode: `iterator()`
- Returverdien til `iterator()` er et objekt som implementerer grensesnittet `Iterator<T>`

Iterator for en klasse

- En iterator er et objekt av en klasse som implementerer `Iterator<T>`
- En klasse som implementerer `Iterator<T>` må implementere to metoder:
  - Den ene sjekker om det gjenstår flere elementer å iterere over
    `public boolean hasNext()`
  - Den andre returnerer det *neste* elementet
    `public T next()`
- En iterator vil som regel implementeres som en *indre klasse* i en beholder
- Vi skal nå gjøre dette konkret for en enkeltlenket liste

Itererbar liste

- Vi kan utvide grensesnittet `Liste<T>` til å være itererbart

```
interface Liste<T> extends Iterable<T> {
    public int antall();
    public void settInnFørst(T elem);
    public void settInnSist(T elem);
    public void fjernFørste();
    public void fjernSiste();
    public T hentFørste();
    public T hentSiste();
    public T hent(int indeks);
}
```

- Merk at nå må alle implementasjoner av `Liste<T>` også implementere `Iterable<T>`

Bruke en iterator

- Å skrive ut alle elementene i en liste kan nå gjøres med en *for-each* løkke

```
1 public void skrivUt(Liste<String> liste) {
2     for (String s : liste) {
3         System.out.println(s);
4     }
5 }
```

- Vi kan også bruke iteratoren *direkte*

```
1 public void skrivUt(Liste<String> liste) {
2     Iterator<String> iterator = liste.iterator();
3     while (iterator.hasNext()) {
4         System.out.println(iterator.next());
5     }
6 }
```

Itererbar enkeltlenket liste

- For å gjøre en listeimplementasjon itererbar må vi implementere metoden `iterator()`
- Den skal *kun returnere* et iteratorobjekt

```
1  class EnkeltLenketListe<T> implements Liste<T> {
2
3      @Override
4      public Iterator<T> iterator() {
5          return new ListeIterator();
6      }
7
8      private class ListeIterator implements Iterator<T> {
9          // ···
10     }
11 }
```

- Vi lar `ListeIterator` være en privat indre klasse

En iterator for enkeltlenkede lister

- Husk hvordan vi kunne løpe gjennom en liste ved å følge neste-pekere
  - En iterator skal kunne gjøre det samme på en *strukturert* måte
  - Altså, via metodene `hasNext()` og `next()`
- For en enkeltlenket liste er det tilstrekkelig å ha en peker til en node
  - Vi kaller denne pekeren `it`
- Iteratoren sin `it` vil starte med å peke på det `første` peker på
- Iteratoren har en neste dersom `it` ikke peker på `null`
- Iteratoren sitt neste elementet er gitt av `it.elem`
  - Når vi kaller `next()` på iteratoren må vi returnere *elementet* `it` peker på
  - Vi må sørge for å oppdatere `it` til å peker på `it` sin neste før metoden returnerer

En iterator for enkeltlenkede lister (kode)

```
1  private class ListeIterator implements Iterator<T> {
2      Node it = første;
3
4      @Override
5      public boolean hasNext() {
6          return it != null;
7      }
8
9      @Override
10     public T next() {
11         T elem = it.elem;
12         it = it.neste;
13         return elem;
14     }
15 }
```

Iteratorer oppsummert

- En klasse eller et grensesnitt kan itereres over dersom de implementerer grensesnittet `Iterable<T>`
- En implementasjon av `Iterable<T>` består av én metode `iterator()`
  - Denne skal kun returnere et objekt av en klasse som implementerer `Iterator<T>`
  - En klasse som implementerer `Iterator<T>` er som regel en indre klasse
- En klasse som implementerer `Iterator<T>` må holde på tilstand slik at
  - Vi kan avgjøre om det er flere elementer som skal løpes over med `hasNext()`
  - Returnere neste element og gjøre klart for neste iterasjon med `next()`
- Merk at vi *ikke går gjennom* `remove()` som er en valgfri metode i grensesnittet `Iterator<T>`

Litt om unntak

Litt om unntak

- En metode i Java kan «kaste et unntak»
- Et unntak vil forårsake at programmet kræsjer med mindre det håndteres eksplisitt
- Når vi kaller på metoder som kan kaste unntak, kan vi prøve og fange opp eventuelle feil

```
1 try { metodeSomKanFeile(); }
2 catch (Exception e) { /* feilhåndtering */ }
```

- Her fanger vi unntak, som er en instans av et objekt `Exception`
- Unntak som kan unngås gjennom vanlig programflyt bør ikke håndteres
  - De bør *unngås* (for eksempel ved å sjekke at en variabel ikke er `null`)
- Vi kan lage våre egne unntak, som gir mer utfyllende informasjon om feilen
  - Dette kan være nyttig for å vite hvilken del av et system som feilet
  - Java støtter *flere* `catch`-blokker, slik at ulike feil kan håndteres forskjellig

To typer unntak

- `Exception` er det mest generelle unntaket
  - Alle unntak arver fra `Exception`
  - Kompilatoren krever at disse håndteres eksplisitt
  - Generelt er dette feil som kan oppstå av grunner som er utenfor vår kontroll (for eksempel at enheten mistet tilgang til internett)
- `RuntimeException`
  - Dette er feil som bør unngås gjennom vanlig programflyt
  - De brukes primært for bedre feilbeskjeder

Et eget unntak

- I Innlevering 3 finner vi et eksempel på et egendefinert unntak

```
class UgyldigListeindeks extends RuntimeException {
    UgyldigListeindeks (int indeks) {
        super("Ugyldig listeindeks: " + indeks);
    }
}
```

- Dette arver fra `RuntimeException` og trenger ikke håndteres eksplisitt
- Det kan kastes ved å bruke `throw`

```
if (indeks < 0 || antall() <= indeks) {
    throw new UgyldigListeindeks(indeks);
}
```