Beholdere

IN1010 – Objektorientert programmering

Lars Tveito

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

Introduksjon

Plan for forelesningen

- Litt generelt om beholdere
- Kort Java sine innebygde beholdere
- Introduksjon til dynamiske arrayer
  - Kjent som `ArrayList` i Java
  - Vi skal forstå hvordan de virker og kunne lage våre egne
- Introduksjon til lenkede lister
  - Kjent som `LinkedList` i Java
  - Vi skal se på flere ulike varianter av lenkede lister
  - Vi skal forstå hvordan de virker og kunne lage våre egne

Beholdere

Beholdere

- En beholder er en datastruktur som brukes til å oppbevare en samling elementer
- Både *lister* og *ordbøker* er eksempler på beholdere som dere kjenner fra før
- Gitt en beholder, vil man vanligvis ha metoder for å
  - Sette inn elementer
  - Fjerne elementer
  - Hente ut elementer
- Beholdere er noe veldig generelt, og de kommer i svært mange varianter som støtter ulike operasjoner
- Som regel er beholdere laget *generiske*
  - Det vil si de kan holde på objekter av en hvilken som helst type
  - Det er temaet for neste uke

Noen eksempler

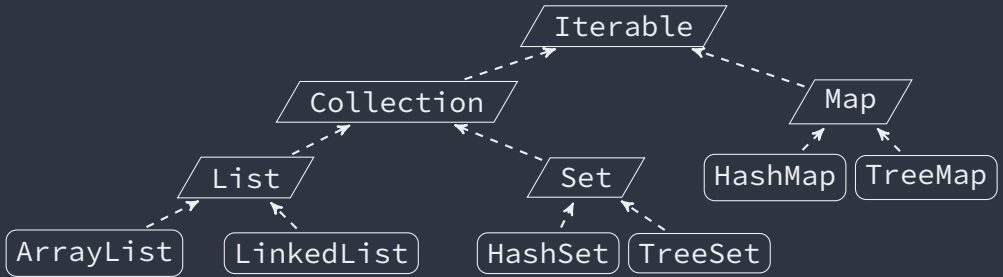
Her er noen eksempler på beholdere, hvor du vil kjenne *noen*, men antageligvis ikke alle, fra før:

```
1 String[]              strengArray       = new String[100];
2 List<String>          strengArrayListe  = new ArrayList<>();
3 List<String>          strengLenketListe = new LinkedList<>();
4 Set<String>           strengMengde      = new HashSet<>();
5 Set<String>           strengTreMengde   = new TreeSet<>();
6 Map<String, String>   strengOrdbok      = new HashMap<>();
7 Map<String, String>   strengTreOrdbok   = new TreeMap<>();
```

- I IN1010 forventer vi at du har dyp kjennskap til arrayer og lister
  - Det vil si at du både kan *bruke* og *lage dine egne*
- Vi forventer at du kjenner til, og kan *bruke,* mengder og ordbøker
  - (Hvordan de implementeres er tema for IN2010)

Grensesnitt for beholdere

- En peker til en beholdere bør som oftest være av en type som er definert av et `interface`
  - Det finnes nesten alltid flere måter å implementere det samme grensesnittet
- Under er et *lite* utsnitt av klassehierarkiet til Java sine innebygde beholdere



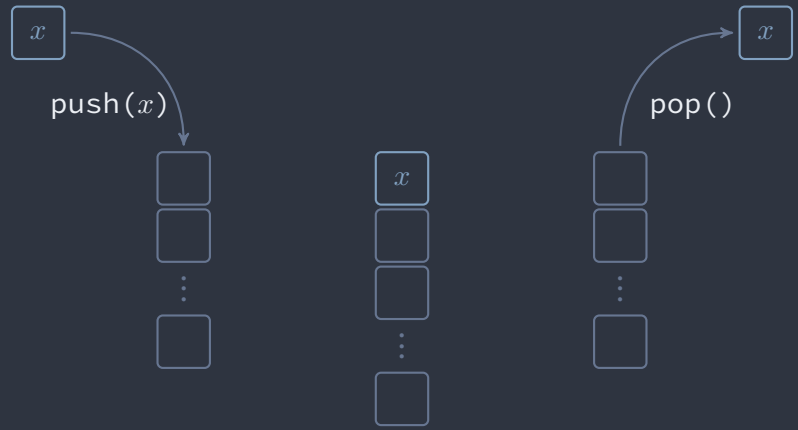
Lister

Lister

- En liste er en *samling* av elementer av *endelig* størrelse i en *bestemt ordning*
- I en liste forventer vi å kunne:
  - Hente ut størrelsen på listen
  - Sette inn et element *på en bestemt posisjon*
  - Fjerne et element på en bestemt posisjon
    - Kanskje også fjerne et element basert på verdi
  - Hente ut elementer på en gitt posisjon
    - Kanskje også finne posisjonen til et gitt element
  - Iterere over elementene i listen *i rekkefølgen* som er bestemt av listen
- Java sitt grensesnitt `List` oppfyller forventningene våre til lister
- Vi skal se på to grensesnitt som avgrenser hvilke operasjoner som støttes
  - Begge kan implementeres som lister

Stack

En *stack* er en liste hvor alle innsettinger og slettinger forekommer på *samme* ende av listen



Kø

En *kø* er en liste hvor alle innsettinger forekommer på den ene enden av listen, og alle slettinger forekommer på den andre enden av listen



- I Java kalles operasjonene `offer()`/`poll()`
  - Kalles også ofte `enqueue()`/`dequeue()`

Et forenklet grensesnitt for lister

- Under er et enkelt grensesnitt for lister av strenger

```
interface StrengListe {
  public int antall();
  public void settInnFørst(String elem);
  public void settInnSist(String elem);
  public void fjernFørste();
  public void fjernSiste();
  public String hentFørste();
  public String hentSiste();
  public String hent(int indeks);
}
```

- Vi skal se på ulike måter å implementere dette grensesnittet

Dynamiske arrayer

Dynamiske arrayer

- I Java er dynamiske arrayer kjent som `ArrayList`
  - `ArrayList` implementerer grensesnittet `List`, og kan brukes som en liste
- Et vanlig array opprettes med en *fast størrelse*
- Et dynamisk array utvides *automatisk*
  - Men automatikken kommer ikke av seg selv
- Nå skal vi implementere et dynamisk array som implementerer grensesnittet `StrengListe`

Klassedeklarasjon

- Vi trenger et heltall `antall` for å holde styr på hvor mange strenger vi har lagret
- Og et array `elementer` som har plass til *minst* én streng
  - Java sin `ArrayList` starter vanligvis med et array med 10 plasser

```
1  class DynamiskStrengArray implements StrengListe {
2
3      private int antall;
4      private String[] elementer;
5
6      DynamiskStrengArray() {
7          antall = 0;
8          elementer = new String[1];
9      }
10 }
```

Finne antall

- For å returnere antall elementer i listen, kan vi returnere `antall`

```
@Override
public int antall() {
    return antall;
}
```

- Merk at metoder og variabler kan samme navn, uten at de kolliderer
- I alle metoder som modifiserer listen må vi sørge for å oppdatere `antall`

Innsetting på slutten

- Når vi setter inn et element, så må vi først sikre at vi har kapasitet
  - Det vil si vi må sørge for at et er plass til *minst* ett element til
  - Vi antar at vi har en metode `sikreKapasitet()` som sørger for dette
- Vi setter verdien på posisjon `antall` til å være strengen ble gitt som argument
- Til slutt inkrementerer vi `antall`

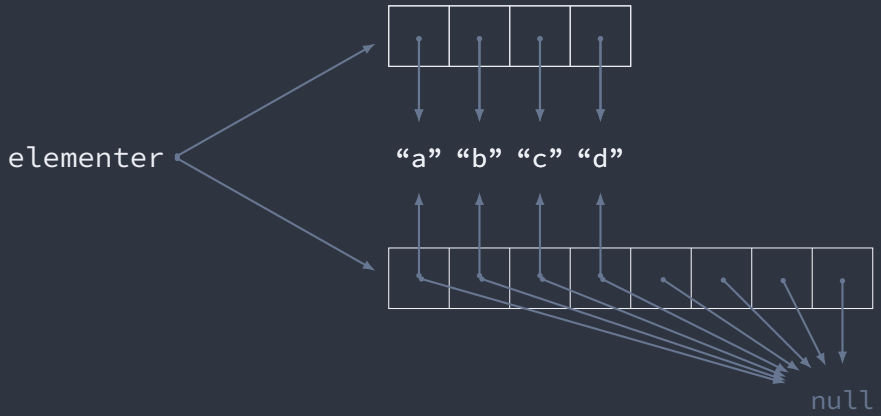
```
1 @Override
2 public void settInnSist(String elem) {
3     sikreKapasitet();
4     elementer[antall] = elem;
5     antall++;
6 }
```

Sikre kapasitet

- Dersom `antall` er *mindre enn* `elementer.length` gjør vi *ingenting*
- Hvis ikke, så oppretter vi et nytt array med dobbelt så mye plass
  - Det nye arrayet fylles opp med verdiene fra `elementer`
- Til slutt setter vi `elementer` til å peke på det nye arrayet
  - Nå har vi plass til flere elementer!

```
1  private void sikreKapasitet() {
2      if (antall < elementer.length) {
3          return;
4      }
5      String[] ny = new String[elementer.length * 2];
6      for (int i = 0; i < antall; i++) {
7          ny[i] = elementer[i];
8      }
9      elementer = ny;
10 }
```

Sikre kapasitet (illustrasjon)



Sletting på slutten

- Når vi fjerner siste element må vi huske å sjekke at `antall` ikke er `0`
  - I så fall gjør vi ingenting
  - Alternativt kunne vi kastet et passende unntak
- Vi senker `antall` og setter elementet på posisjon `antall` til `null`

```
1  @Override
2  public void fjernSiste() {
3      if (antall == 0) {
4          return;
5      }
6      antall--;
7      elementer[antall] = null;
8  }
```

Hente første og siste

- Å hente første er å slå opp på indeks `0`
- Å hente siste er å slå opp på indeks `antall - 1`
- Vi returnerer `null` dersom arrayet er tomt
  - Her kunne vi også kastet et passende unntak

```java
@Override
public String hentFørste() {
    if (antall == 0) {
        return null;
    }
    return elementer[0];
}
```

```java
@Override
public String hentSiste() {
    if (antall == 0) {
        return null;
    }
    return elementer[antall - 1];
}
```

Hente på indeks

- Å hente på indeks gjøres ved å slå opp på indeksen i `elementer`
- Her kaster vi `IndexOutOfBoundsException` dersom indeksen er negativ eller større (eller lik) `antall`.

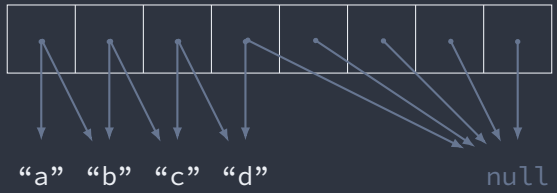
```
1  @Override
2  public String hent(int indeks) {
3      if (indeks < 0 || antall() <= indeks) {
4          throw new IndexOutOfBoundsException();
5      }
6      return elementer[indeks];
7  }
```

Sett inn først

- Innsetting først i et dynamisk array er litt mer knot
- Antageligvis ligger det allerede noe på posisjon `0`
- Derfor må vi flytte alle elementer en posisjon frem

```java
@Override
public void settInnFørst(String elem) {
    sikreKapasitet();
    for (int i = antall; 0 < i; i--) {
        elementer[i] = elementer[i - 1];
    }
    elementer[0] = elem;
    antall++;
}
```

Sett inn først (illustrasjon)



Fjern først

- Tilsvarende er fjerning først litt knot
- Her må vi flytte alle gjenværende elementer en posisjon tilbake
- Vi må også huske å sette siste posisjon til `null`

```
@Override
public void fjernFørste() {
    if (antall == 0) {
        return;
    }
    for (int i = 0; i < antall - 1; i++) {
        elementer[i] = elementer[i + 1];
    }
    antall--;
    elementer[antall] = null;
}
```

Noder

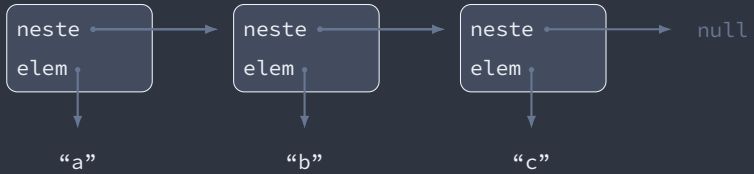
Noder

- En annen tilnærming til lister er å koble sammen *objekter* i en *lenkestruktur*
- En *node* er et objekt, som holder på et element, og har referanser til andre noder
- Dette er et eksempel på en node-klasse

```
1  class Node {
2      String elem;
3      Node neste;
4
5      Node(String elem) {
6          this.elem = elem;
7      }
8  }
```

- Den har ingen metoder utenom konstruktøren

Et lite eksempel med noder

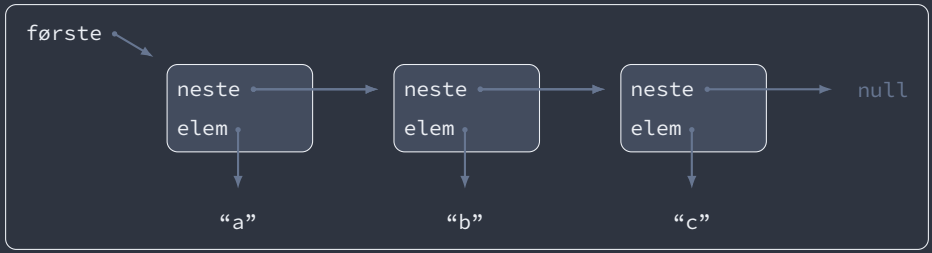



```
Node a = new Node("a");
a.neste = new Node("b");
a.neste.neste = new Node("c");
// ...denne leken kan vi fortsette med så lenge vi ønsker
```

Enkeltlenket liste

Enkeltlenket liste

- Ved å knytte noder sammen med *neste*-pekere får vi en *enkeltlenket liste*
- Vi kan lage lister som objekter som har en peker til den første noden
- `null` representerer den tomme listen



Klassedeklarasjon

```
1  class EnkeltLenketStrengListe implements StrengListe {
2
3      private class Node {
4          String elem;
5          Node neste;
6
7          Node(String elem, Node neste) {
8              this.elem = elem;
9              this.neste = neste;
10         }
11     }
12
13     private Node første;
14 }
```

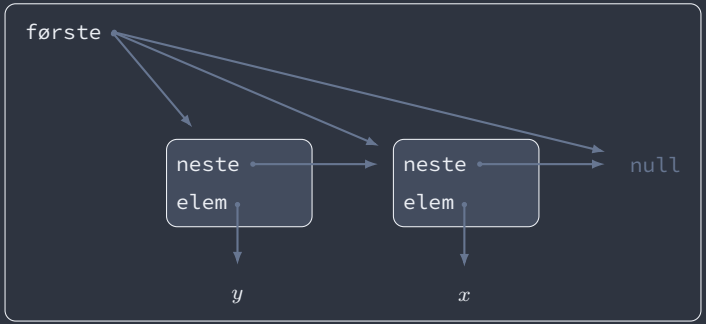
- Vi lar `Node` være en *indre* klasse
- Merk at vi her setter `neste`-pekeren i nodens konstruktør

Sett inn først

- Når vi setter inn først i en enkeltlenket liste må vi:
  - Lage en ny node
  - La den nye noden sin neste peke på det `første` peker på
  - Sette `første` til å peke på den nye noden

```
1  @Override
2  public void settInnFørst(String elem) {
3      Node ny = new Node(elem, første);
4      første = ny;
5  }
```

Sett inn først (illustrasjon)



Fjern første

- For å fjerne første element må vi sette `første` til å peke på `første.neste`
  - Nå er det ikke lenger noen som peker på noden `første` pekte på
  - Da «forsvinner» den ved neste søppeltømming (garbage collection)
- Igjen gjør vi ingenting dersom `første` er `null`

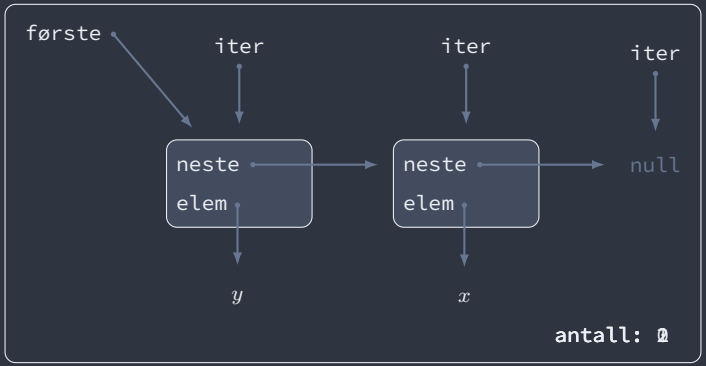
```
1  @Override
2  public void fjernFørste() {
3      if (første != null) {
4          første = første.neste;
5      }
6  }
```

Finne antall

- Vi har «glemt» å holde styr på antall noder som er lagret
- Da må vi løpe gjennom listen for å finne antall noder
  - Det gjøres ved å opprette en nodepeker `iter` som initialiseres til å peke på det `første` peker på
  - Deretter følge `neste`-pekere frem til vi når `null`
  - Vi følger en `neste`-peker ved å si `iter = iter.neste`

```
1  @Override public int antall() {
2      int antall = 0;
3      Node iter = første;
4      while (iter != null) {
5          antall++;
6          iter = iter.neste;
7      }
8      return antall;
9  }
```

Finne antall (illustrasjon)



Finne antall litt lurere

- Å gå gjennom hele listen for å finne antallet er *svært ineffektivt*
  - Dersom dette er en operasjon som skal brukes bør listen holde oversikt over antallet i en instansvariabel
  - Det gjør alle metodene som endrer på listen *litt* mer kompliserte

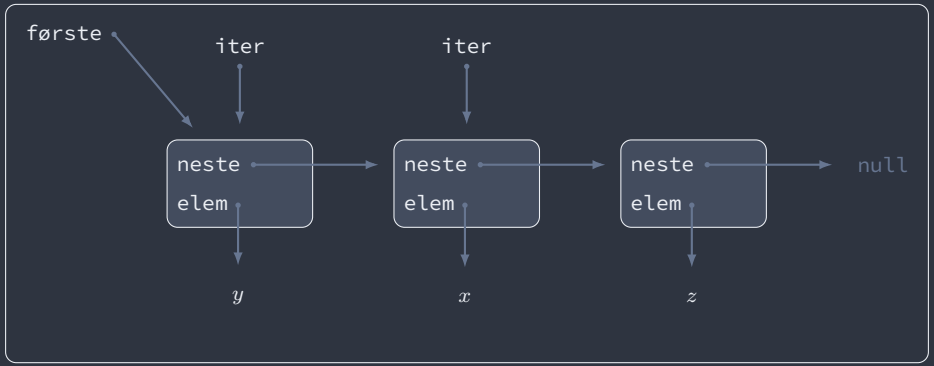
```
1  class EnkeltLenketStrengListeMedAntall implements StrengListe {
2
3      private class Node { /* ··· */ }
4
5      private Node første;
6      private int antall;
7  }
```

Sett inn sist

For å sette inn sist i en enkeltlenket liste må vi finne *siste* node og sette den sin neste til en ny node som holder på elementet

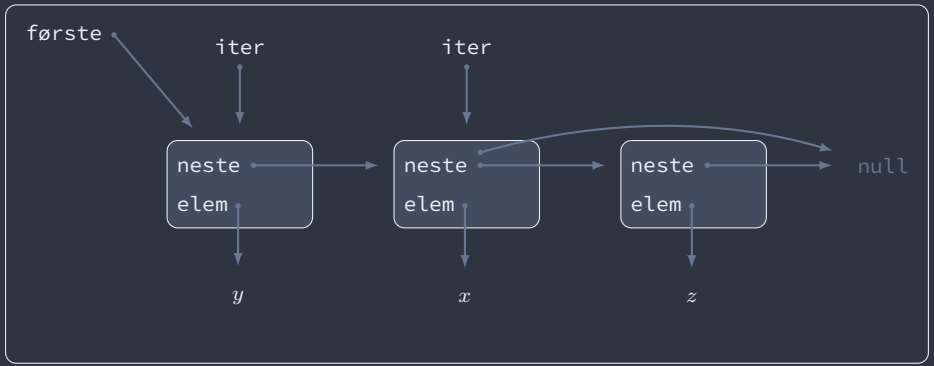
```
@Override
public void settInnSist(String elem) {
    Node ny = new Node(elem, null);
    if (første == null) {
        første = ny;
    } else {
        Node iter = første;
        while (iter.neste != null) {
            iter = iter.neste;
        }
        iter.neste = ny;
    }
    antall++;
}
```

Sette inn sist (illustrasjon)



Fjerne sist (illustrasjon)

For å fjerne siste i en enkeltlenket liste må vi finne *nest siste* node og sette den til å peke på sin nestes neste!



Lurere måte å sette inn sist

- Å fjerne siste *er* ugunstig med enkeltlenkede lister
- Å sette inn kan gjøres enklere ved å ha en peker til siste node

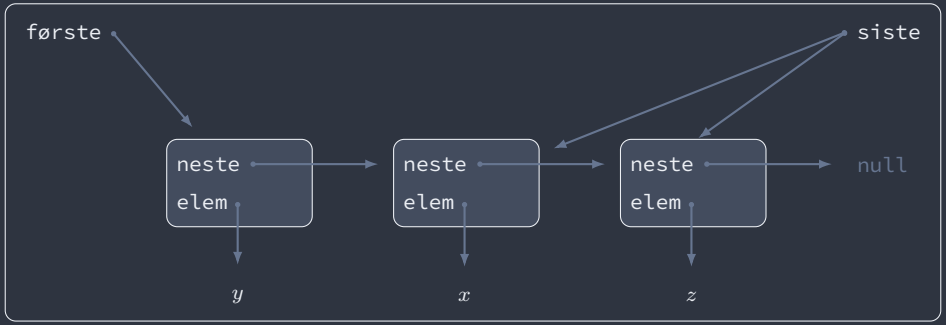
```
1  class EnkeltLenketStrengListeMedSiste implements StrengListe {
2
3      private class Node { /* ··· */ }
4
5      private Node første;
6      private Node siste;
7      private int antall;
8  }
```

Innsetting med peker til siste

- Vi skal sette inn på slutten av liste *med en* peker til siste node
- Vi oppretter en ny node med som har `null` som sin neste
- Hvis listen er tom må både `første` og `siste` peke på den nye noden
- Hvis ikke, settes `siste` *og* siste sin neste til å peke på den nye noden

```
1  @Override
2  public void settInnSist(String elem) {
3      Node ny = new Node(elem, null);
4      if (første == null) {
5          første = ny;
6          siste = ny;
7      } else {
8          siste.neste = ny;
9          siste = ny;
10     }
11     antall++;
12 }
```

Innsetting med peker til siste (illustrasjon)



En lurere måte å fjerne siste (neste uke)

- Sistepekeren hjelper oss ikke med å *fjerne* siste node
- Det er fordi vi må oppdatere den *nest* siste noden for å slette på slutten
- Hvis vi hadde hatt en peker tilbake til forrige node ville alt vært så mye greiere!
- Det er en fortelling for neste uke