Kapittel 11:
Implementering av dynamiske datastrukturer

Redigert av:

Khalid Azim Mughal (`khalid@ii.uib.no`)

Kilde:

Java som første programmeringsspråk (3. utgave)

Khalid Azim Mughal, Torill Hamre, Rolf W. Rasmussen
Cappelen Akademisk Forlag, 2006.
ISBN: 82-02-24554-0
`http://www.ii.uib.no/~khalid/jfps3u/`
(NB! Boken dekker opptil Java 6, men notatene er oppdatert til Java 7.)

Emneoversikt

Kjedete lister

- innsetting
- sletting
- oppslag

Andre ADT'er:

- Stabler
- Køer

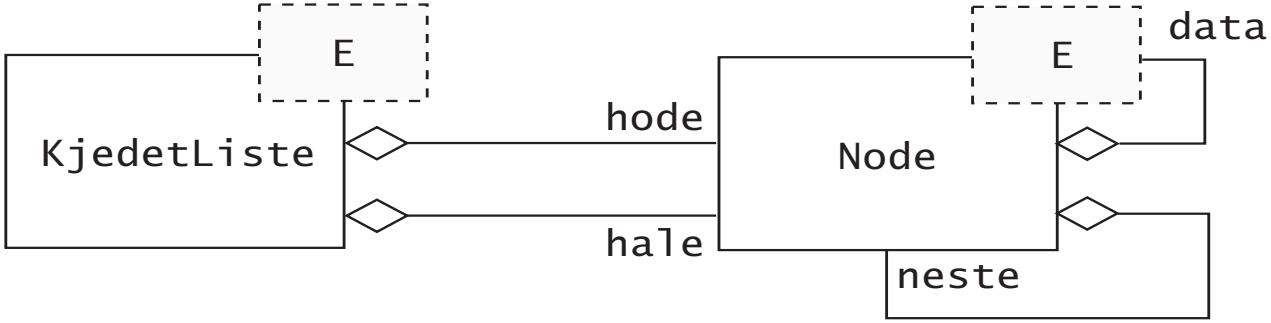
Kjedete lister

- En *enkel kjedet liste* er en *lineær* samling av objekter, kalt *noder*.
- En node er et eksempel på et objekt som har en referanse til et objekt av samme type, dvs. er *selvrefererende*.
- Klassen `Node<E>` definerer to feltvariabler:
  - Referansen `data` som refererer til et vilkårlig objekt.
  - Referansen `neste` som refererer til en node.
- Nodene kan *kjedes* sammen v.h.a. `neste`-felt til å danne en kjedet liste.

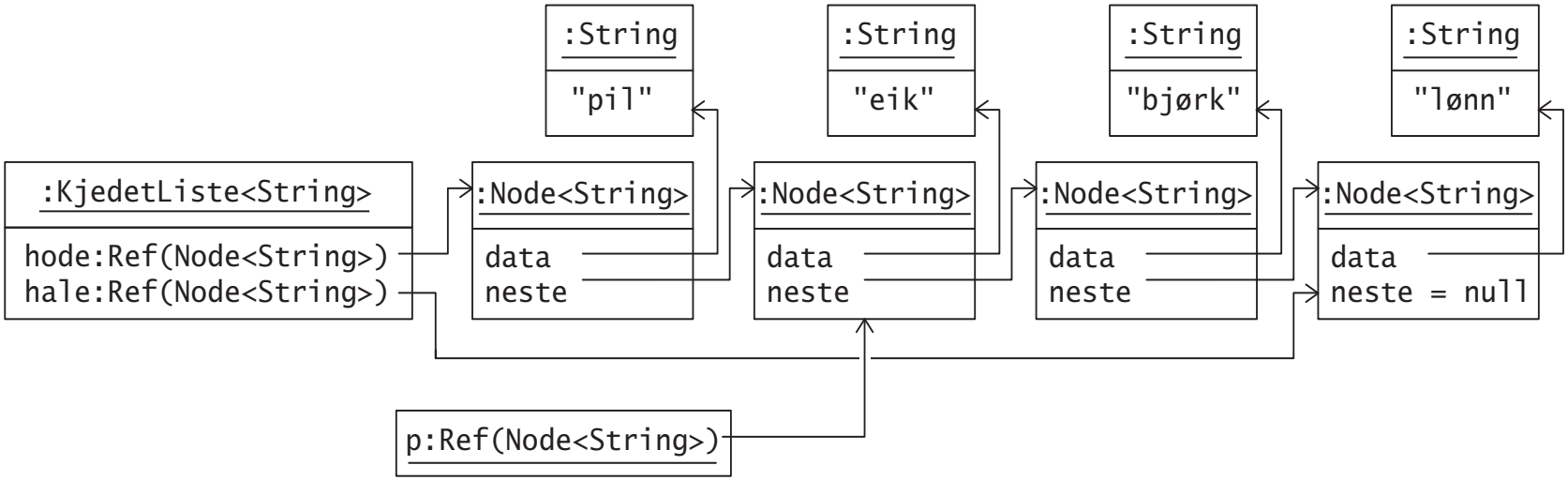
Klassen Node

```
/** En node inneholder data, og referanse til neste node. */
public class Node<E> {

  /** Data i noden. */
  private E data;
  /** Referanse til neste node. */
  private Node<E> neste;

  /** Konstruktør */
  public Node(E data_obj, Node<E> nodeRef) {
    data = data_obj;
    neste = nodeRef;
  }

  /** Mutatorer */
  public void settData(E obj) { data = obj; }
  public void settNeste(Node<E> node) { neste = node; }

  /** Selektorer */
  public E hentData() { return data; }
  public Node<E> hentNeste() { return neste; }
}
```

Klassediagram for enkel kjedet liste (Figur 9.4)



En konkret kjedet liste med strenger



- Å sette inn i og hente data fra en node:

```
String str = hode.hentData(); // Henter data fra 1. node.
p.settData("ask");  // Data i noden betegnet av p endres til "ask".
```

- Finne nodens etterfølger:

```
Node<String> etterfølger = p.hentNeste(); // Henter etterfølger til p.
```

- Sette etterfølgeren til en noden:

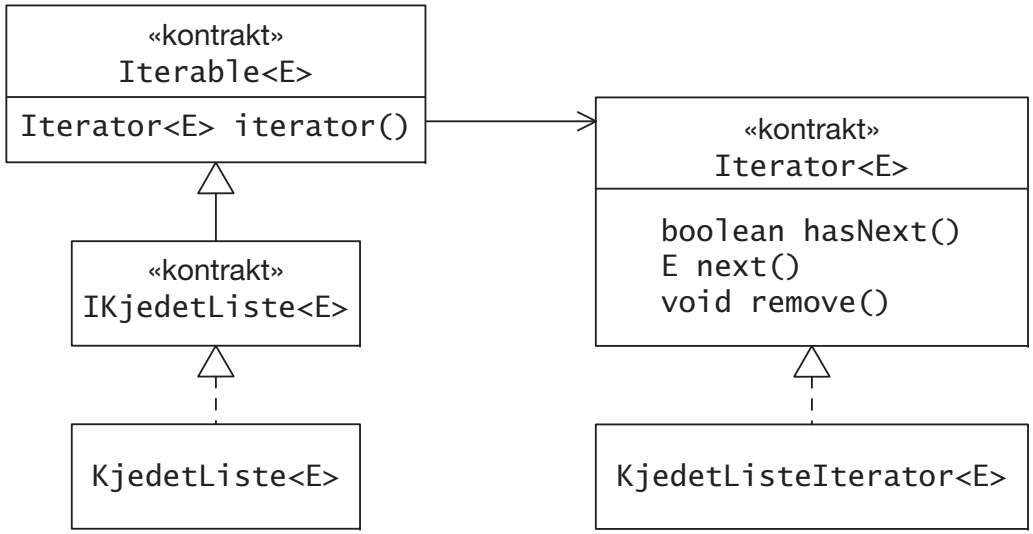
```
p.settNeste(q);    // q settes som etterfølgeren til p.
```

- Flytte referansen `p` slik at den betegner etterfølgeren:

```
p = p.hentNeste(); // referansen p betegner nå etterfølgeren.
```

Kontrakten IKjedetListe<E> (Program 11.2, Figur 11.3)

```
interface IKjedetListe<E> extends Iterable<E> {
  // Mutatorer
  void innsettForan(E data_obj);
  void innsettBak(E data_obj);
  E slettForan();
  E slettBak();
  boolean slett(E data_obj);

  // Selektorer
  boolean erTom();
  int antallNoder();
  Node<E> første();
  Node<E> siste();
  Node<E> finn(E data_obj);

  // Andre nyttige metoder
  void listeTilTabell(E[] tab);
}
```



(a) Implementering av kjedete lister (b) Implementering av iterator for kjedete lister

Klassen KjedetListe<E>

```
public class KjedetListe<E> implements IKjedetListe<E> {

    /** Referanse til første node i kjeden. */
    private Node<E> hode;
    /** Referanse til siste node i kjeden. */
    private Node<E> hale;
    /** Antall noder i kjeden. */
    private int antall;
    // ...
}
```

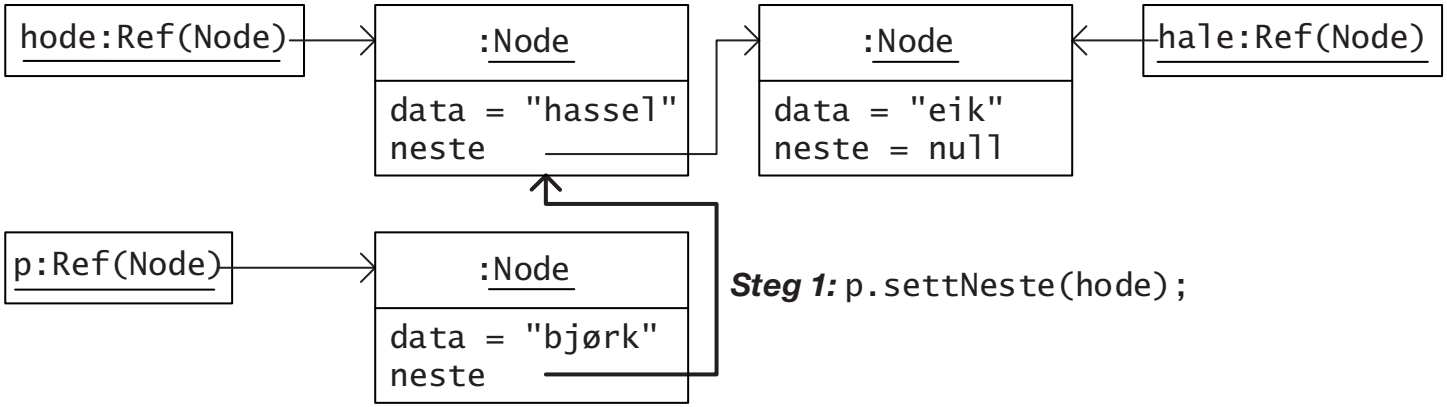
Innsetting foran i en kjedet liste (Program 9.7)

Hvis listen er tom:
Hodet og halen settes til å referere til den nye noden.
ellers:
Steg 1: Sett den nye nodens etterfølger lik hodet.
Steg 2: Sett hodet lik til å referere til den nye noden.

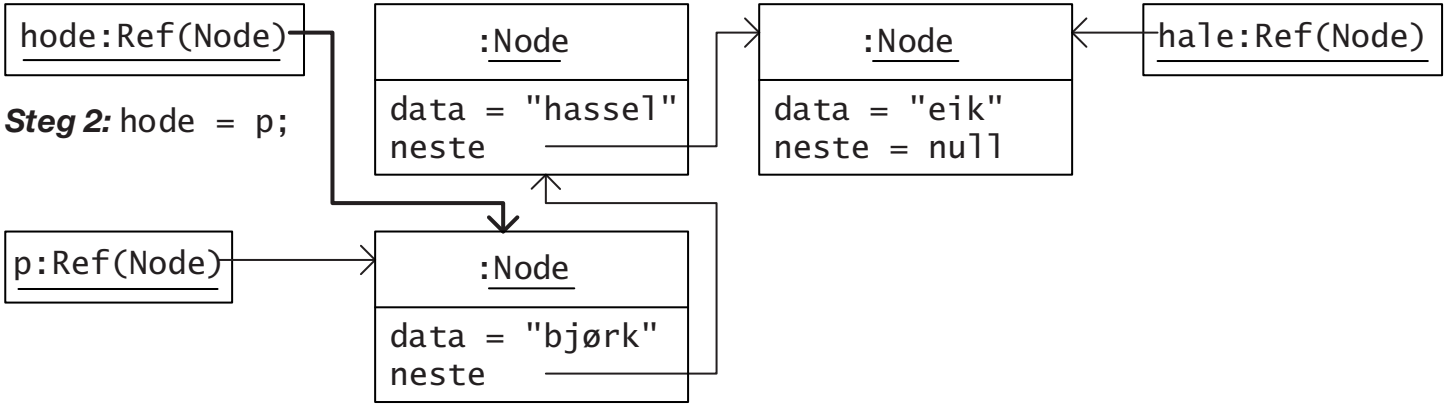
```
public void innsettForan(E data_obj) {                    // (1)
  Node<E> p = new Node<>(data_obj, null);
  if (erTom())
    hode = hale = p;
  else { // hode = new Node(data_obj, hode);
    p.settNeste(hode); // Steg 1
    hode = p;          // Steg 2
  }
  antall++;
}
```

- Det er viktig at de to stegene utføres i riktig rekkefølge (Figur 11.4).

Innsett en node foran i en kjedet liste (Figur 11.4)



(b) Steg 1 i innsetting foran i listen



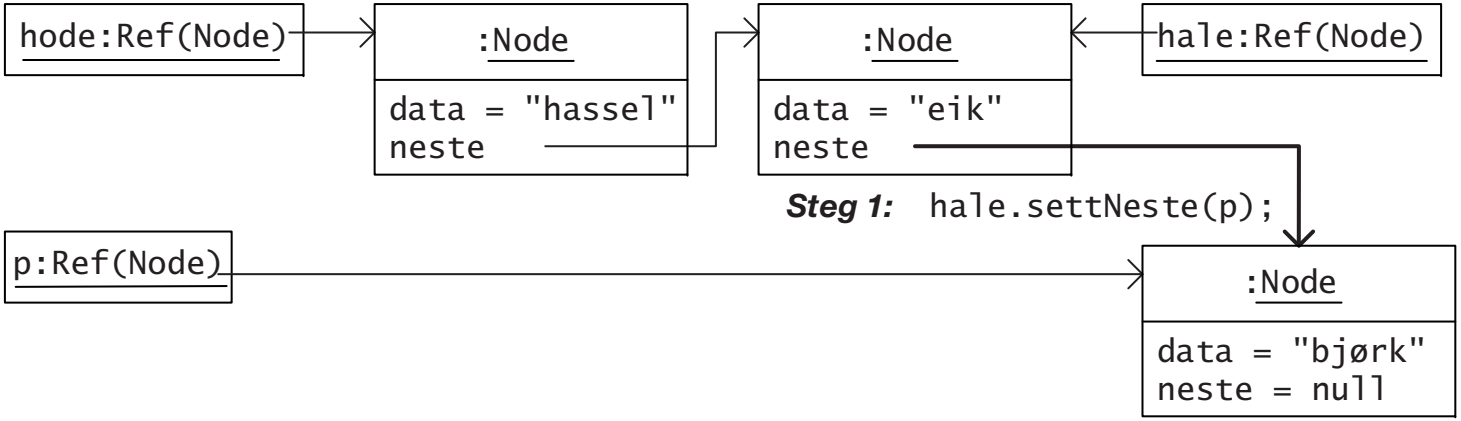
(c) Steg 2 i innsetting foran i listen

Innsetting bak i en kjedet liste

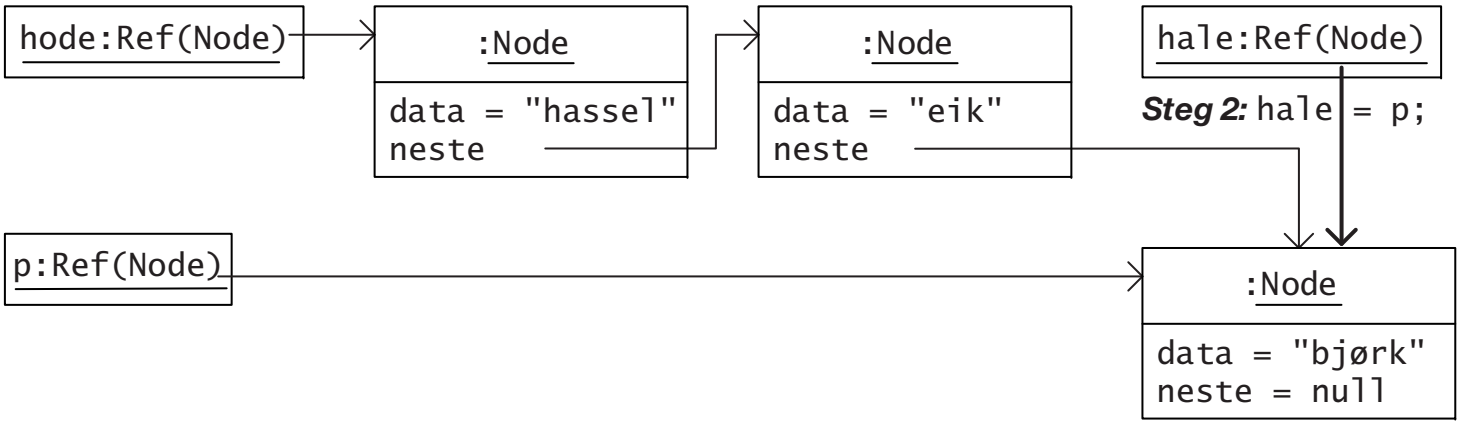
Hvis listen er tom:
Hodet og halen settes til å referere til den nye noden.
ellers:
Steg 1: Sett halens etterfølger til å referere til den nye noden.
Steg 2: Sett halen til å referere til den nye noden.

```
public void innsettBak(E data_obj) {                        // (2)
  Node<E> p = new Node<>(data_obj, null);
  if (erTom())
    hode = hale = p;
  else {
    hale.settNeste(p); // Steg 1
    hale = p;          // Steg 2
  }
  antall++;
}
```

Innsett en node bak i en kjedet liste (Figur 9.7b og c)



(b) Steg 1 i innsetting bak i listen



(c) Steg 2 i innsetting bak i listen

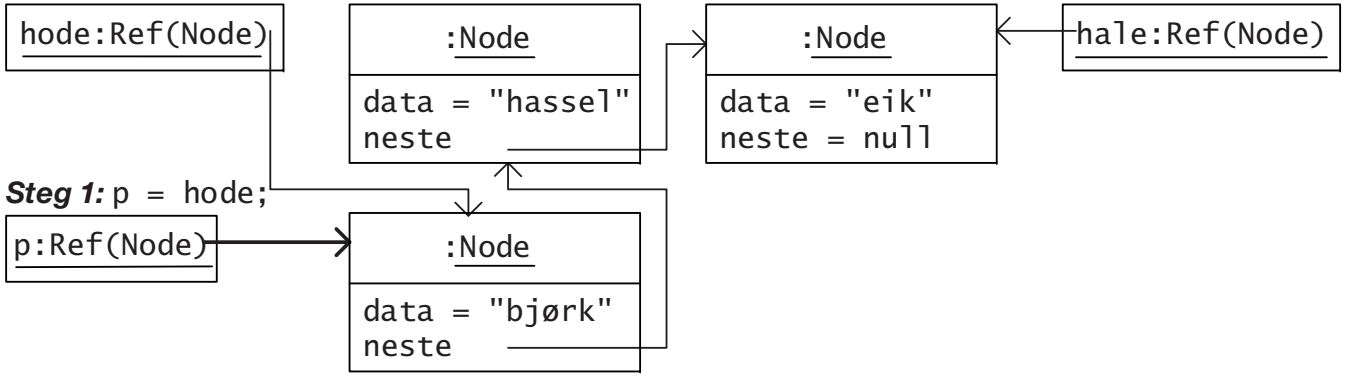
Sletting foran i en kjedet liste

Steg 1: La p betegne første node i listen, dvs. samme node som hodet.

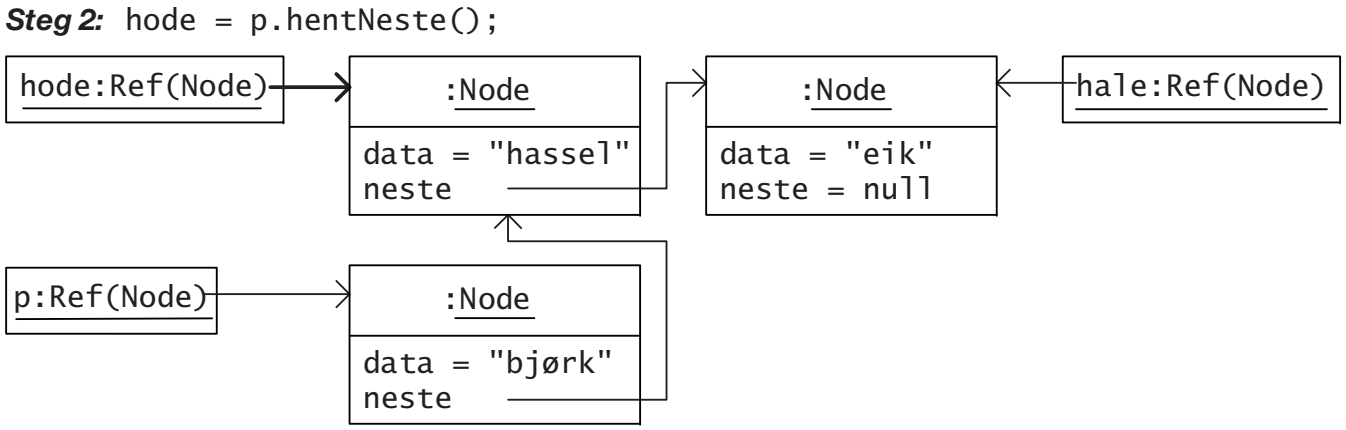
Hvis hodet og halen referere til samme node, m.a.o. bare en node i listen:
Både hodet og halen settes til null.
ellers:
Steg 2: Sett hodet til å referere til etterfølgeren av første node angitt av p

```
public E slettForan() {                                  // (3)
  assert antall > 0 : "Listen kan ikke være tom.";
  Node<E> p = hode;       // Steg 1
  if (hode == hale)
    hode = hale = null;   // Bare en node i listen
  else
    hode = p.hentNeste(); // Steg 2
  antall--;
  return p.hentData();
}
```

Slett node foran i en kjedet liste (Figur 11.6b og c)



(b) Steg 1 i sletting foran i listen

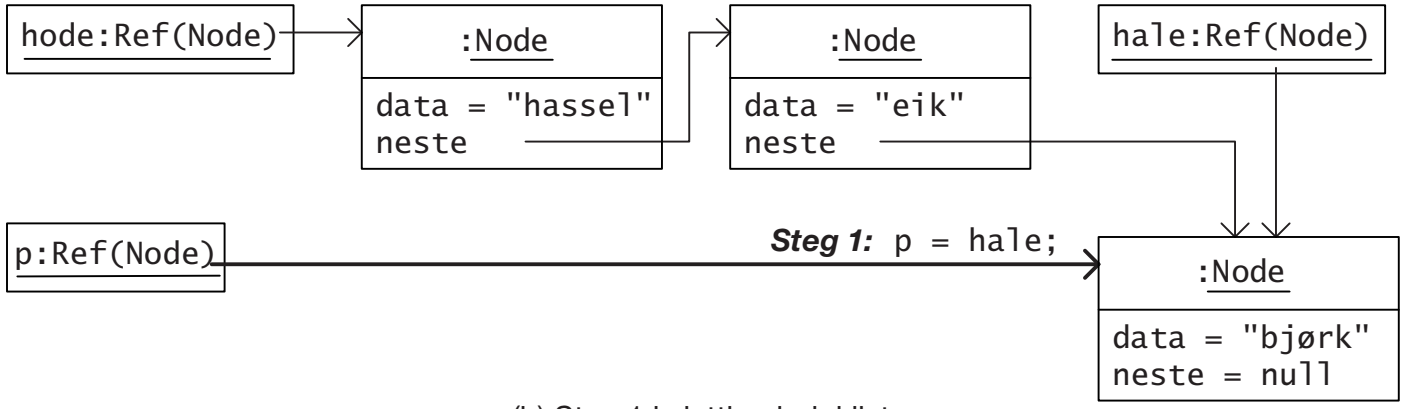


(c) Steg 2 i sletting foran i listen

Sletting bak i en kjedet liste (Program 9.7)

- Vi lar referanse `p` betegne noden som skal slettes (Figur 11.6, steg 1):

```
p = hale;                          // Steg 1
```



(b) Steg 1 i sletting bak i listen

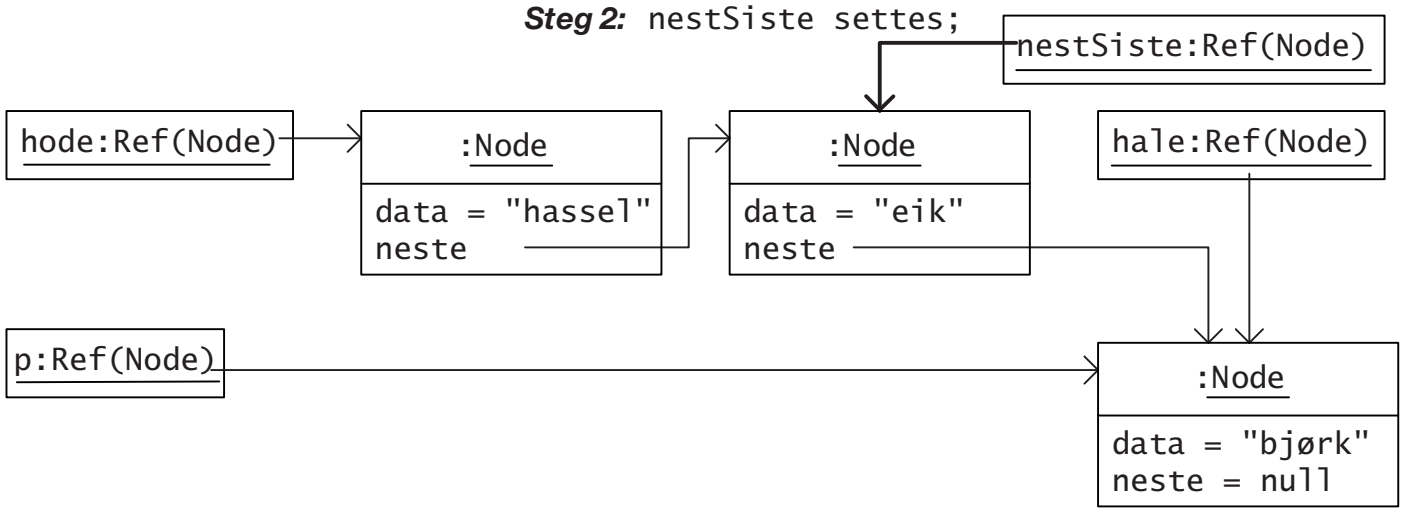
- Sletting av en node inne i en liste krever kjennskap til forgjengeren til noden som skal slettes.
  - For å finne den nest siste noden er vi nødt til å traversere listen fra hodet:

    La referansen nestSiste betegne første node i listen, dvs. samme node som hodet.

    Gjenta mens ikke kommet til nest siste node:
    Flytt referansen nestSiste til etterfølgeren.

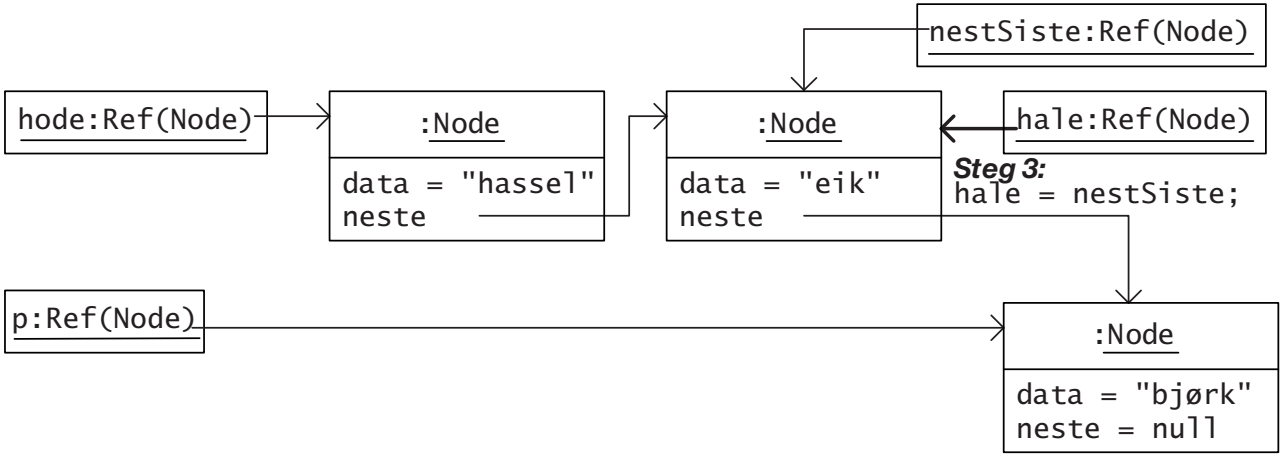
– Kildekode for å finne den nest siste noden (Figur 11.7, steg 2):

```
Node<E> nestSiste = hode;          // Steg 2
while (nestSiste.hentNeste() != hale)
  nestSiste = nestSiste.hentNeste();
```

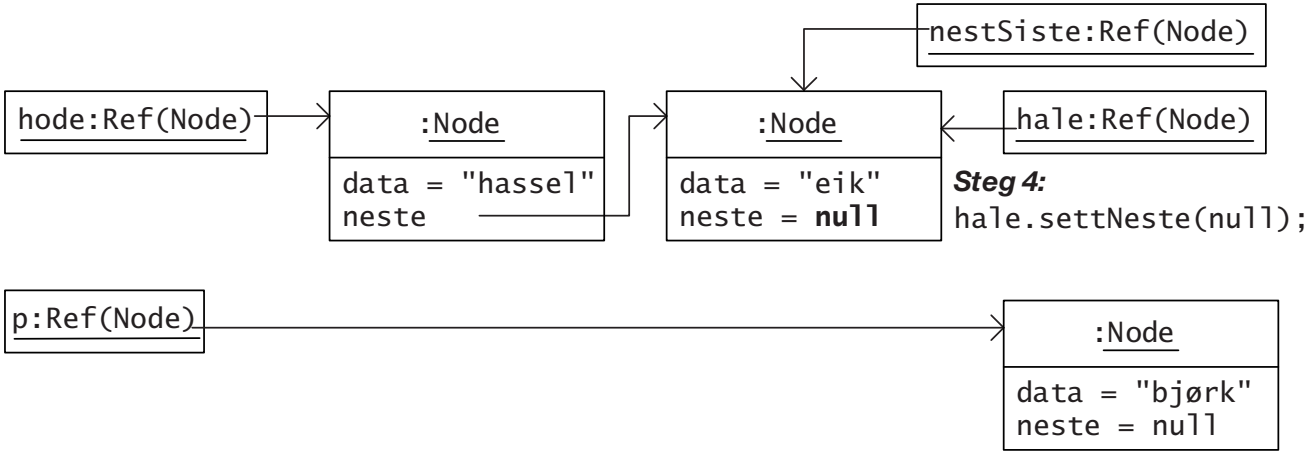


(c) Steg 2 i sletting bak i listen

Slett en node bak i en kjedet liste (Figur 11.8, stegene 3 og 4)



(d) Steg 3 i sletting bak i listen



(e) Steg 4 i sletting bak i listen

```
public E slettBak() {                                    // (4)
  assert antall > 0 : "Listen kan ikke være tom.";
  Node<E> p = hale;           // Steg 1
  if (hode == hale)
    hode = hale = null;       // Bare en node i listen
  else {
    Node<E> nestSiste = hode; // Steg 2
    while (nestSiste.hentNeste() != hale)
      nestSiste = nestSiste.hentNeste();

    hale = nestSiste;         // Steg 3
    hale.settNeste(null);     // Steg 4
  }
  antall--;
  return p.hentData();
}
```

Sletting inne i en kjedet liste (Program 9.7)

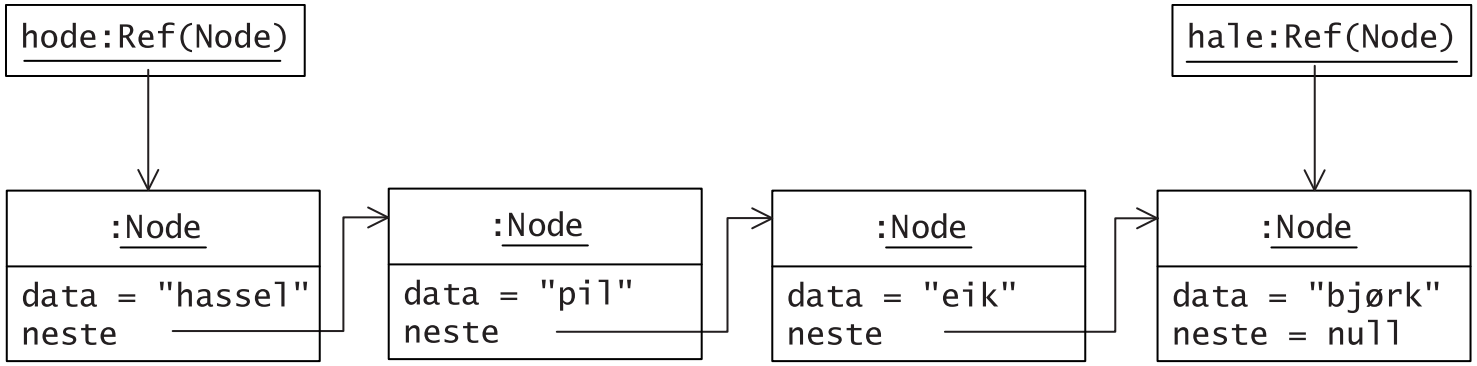
- Finn noden som skal slettes og dens forgjenger (steg 1):

Gjenta mens ikke slutt på listen og ikke funnet noden som skal slettes:
    Hvis inneværende node i listen er den som skal slettes:
        Noden er funnet, og dermed forgjengeren.
    ellers:
        Flytt referansene for inneværende node og forgjengeren.

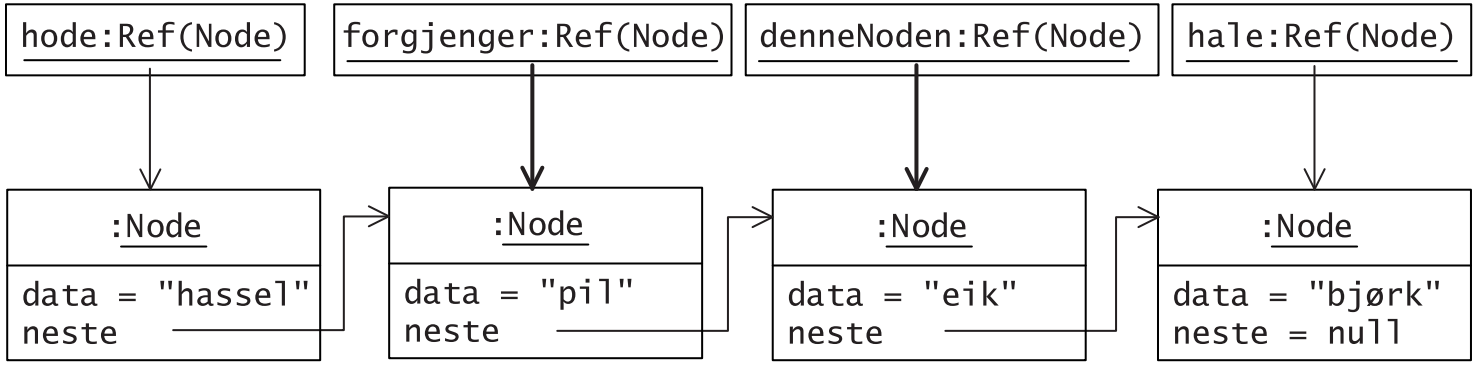
- Kilde for å finne noden som skal slettes og dens forgjenger:

```
Node<E> denneNoden = hode;
Node<E> forgjenger = null;
boolean funnet = false;
while (denneNoden != null && !funnet) {
  if (denneNoden.hentData().equals(data_obj)) {
    funnet = true;
  } else {
    // Ikke funnet. Oppdater forgjenger og denneNoden referanser.
    forgjenger = denneNoden;
    denneNoden = denneNoden.hentNeste();
  }
}
```

Sletting av en node inne i listen (Figur 11.9, steg 1)



(a) Før sletting av en node inne i listen



Steg 1: Finn noden som skal slettes,
og dens forgjenger;

(b) Steg 1 i sletting av en node inne i listen

- Det er tre muligheter å vurdere for å slette en node som finnes i listen:
  - Noden er *første node* i listen (`forgjenger == null`).
    Metoden `slettForan()` kalles til å slette den.

    ```
    slettForan();                // Bruker slettForan()-metoden i klassen
    ```

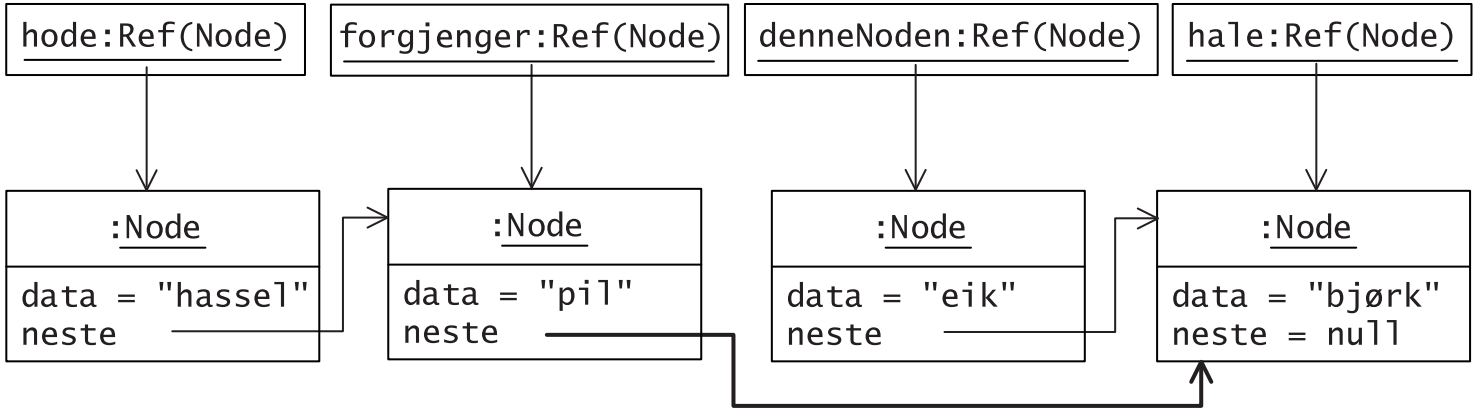
  - Noden er *siste node* i listen (`denneNoden.hentNeste() == null`).
    Halen og siste node oppdateres:

    ```
    hale = forgjenger;
    hale.settNeste(null);
    ```

  - Noden er *inne i* listen.
    Etterfølgeren til noden som skal slettes, blir nå etterfølgeren til forgjengeren (steg 2):

    ```
    forgjenger.settNeste(denneNoden.hentNeste());
    ```

Sletting av en node inne i listen (Figur 11.9, steg 2)



Steg 2: forgjenger.settNeste(denneNoden.hentNeste());

(c) Steg 2 i sletting av en node inne i listen

Finne data i en kjedet liste

- Metoden bruker `equals()`-metoden til å foreta objektlikhet i listen.
- Metoden returnerer noden som inneholder objektet som ble funnet.

```
public Node<E> finn(E data_obj) {
  // Traverser gjennom kjeden fra hode
  Node<E> denneNoden = hode;
  while (denneNoden != null) {
    if (denneNoden.hentData().equals(data_obj))
      return denneNoden;                    // Funnet.
    denneNoden = denneNoden.hentNeste();
  }
  return null;                              // Ikke funnet.
}
```

Iterator for kjedet liste.

- Klassen `KjedetListe<E>` kontrakten `IKjedetListe<E>`, og dermed også indirekte kontrakten `Iterable<E>` (Se Figur 11.3).
- Klassen `KjedetListeIterator` implementerer en iterator for klassen `KjedetListe`:

```
public class KjedetListeIterator<E> implements Iterator<E> {
  private Node<E> denneNoden;

  public KjedetListeIterator(IKjedetListe<E> liste) {
    denneNoden = liste.første();
  }

  public boolean hasNext() { return denneNoden != null; }

  public E next() {
    E data = denneNoden.hentData();
    denneNoden = denneNoden.hentNeste();
    return data;
  }

  public void remove() { throw new UnsupportedOperationException(); }
}
```

Konvertere en kjedet liste til en tabell

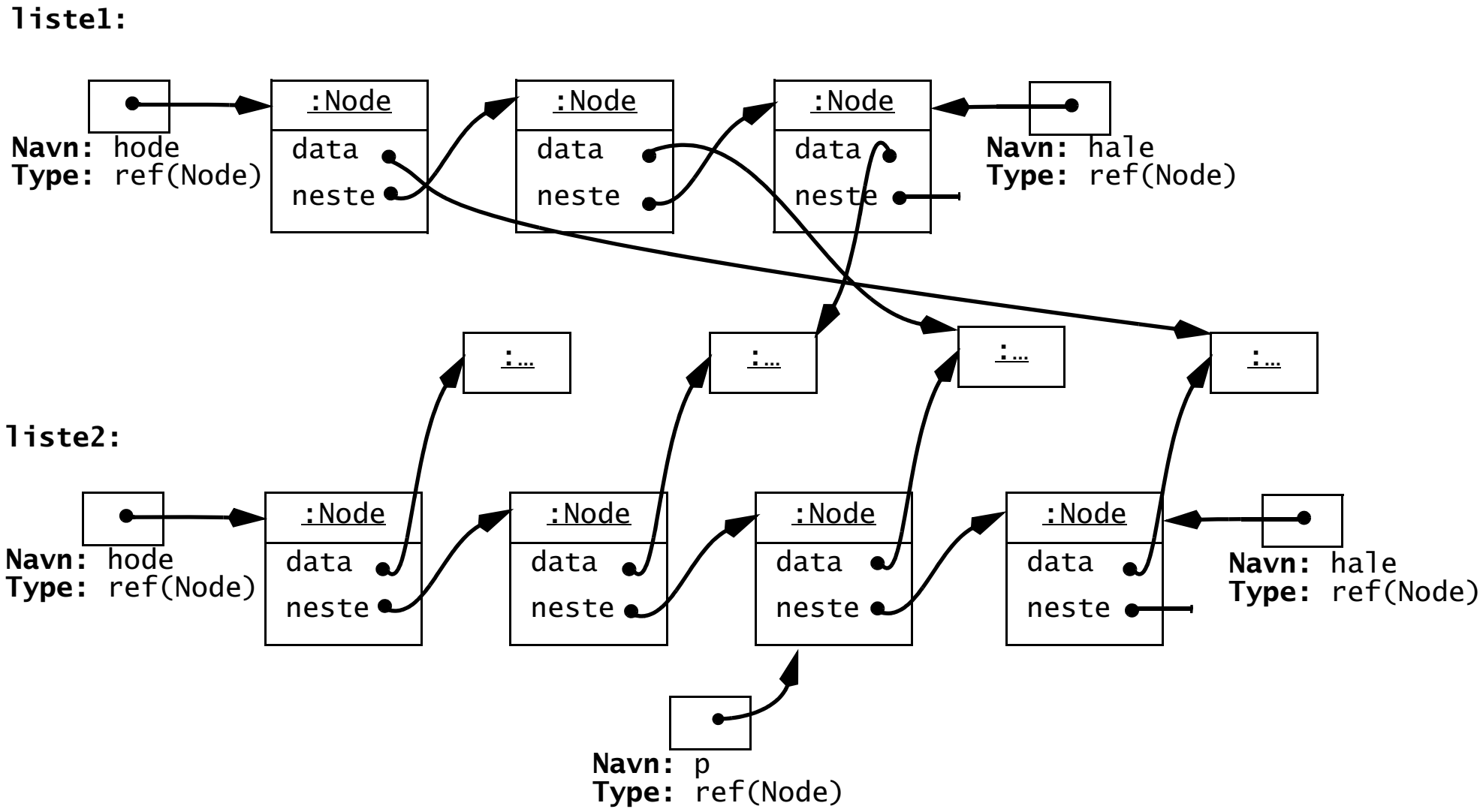
- Metoden fyller angitt tabell med elementer fra listen.

```
public void listeTilTabell(E[] tabell) {                // (7)
  assert tabell.length == antall : "Ikke riktig tabellstørrelse.";
  int i = 0;
  for (E data : this)
    tabell[i++] = data;
}
```

Eksempel: Konvertering av kjedet liste til tabell

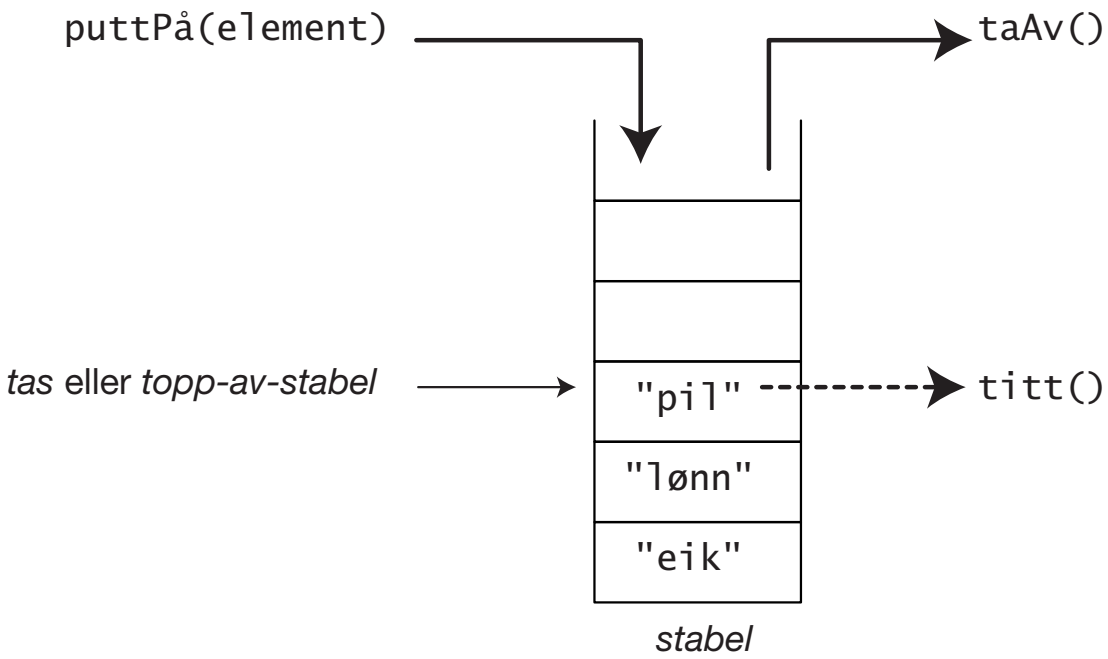
```
public class ListeTilTabellDemo {

  public static void main(String[] args) {
    KjedetListe<Character> lst = new KjedetListe<>();
    lst.innsettForan('a'); lst.innsettForan('b');
    lst.innsettForan('b'); lst.innsettForan('a');
    for (char tegn : lst)
      System.out.print(tegn);
    System.out.println();

    Character[] tab = new Character[lst.antallNoder()];
    lst.listeTilTabell(tab);
    System.out.println(tab.length);
    for (char tegn : tab)
      System.out.print(tegn);
    System.out.println();
  }
}
```

Data kan deles mellom kjedete lister



Datastrukturen stabel

- En *stabel* er en *SIFU* (*sist-inn, først-ut*)-datastruktur (Figur 9.12).



Stabeloperasjoner

Stabeloperasjon	Beskrivelse
`puttPå(element)`	Putter `element` på toppen av stabelen (eng. *push()*).
`taAv()`	Fjerner og returnerer det første elementet på toppen av stabelen (eng. *pop()*).
`titt()`	Returnerer det første elementet på toppen av stabelen uten å fjerne det fra stabelen (eng. *peek()*).
`tom()`	Returnerer sann eller usann avhengig av om stabelen er tom eller ikke (eng. *empty()*).

Stabelimplementering ved aggregering

- Stabeloperasjoner blir oversatt til kjedet liste-operasjoner.
- I denne implementeringen kan ikke en klient bryte abstraksjonen som en stabel representerer.
- Klassen `java.util.Stack<E>` gir en annen implementering av stabler.

```
class StabelVedAggregering<E> {
  // Felt
  private KjedetListe<E> stabelListe;             // (1)

  // Konstruktør
  public StabelVedAggregering() {
    stabelListe = new KjedetListe<>();            // (2)
  }

  // Instansemetoder
  public void puttPå(E data) {                    // (3)
    stabelListe.innsettForan(data);
  }
```

```
  public E taAv() {                                  // (4)
    if (tom()) return null;
    else return stabelListe.slettForan();
  }
  public E titt() {                                  // (5)
    if (tom()) return null;
    Node<E> førsteNode = stabelListe.første();
    return førsteNode.hentData();
  }
  public boolean tom() {
    return stabelListe.erTom();                      // (6)
  }
}
```

Klient som bruker en stabel (StabelKlient.java)

```
public class StabelKlient {
  public static void main(String[] args) {
    // (1) Kontroller om det er noen kommandolinjeargumenter.
    if (args.length == 0) {
      System.out.println("Bruk: java StabelKlient <argumentliste>");
      return;
    }

    // (2) Opprett en stabel.
    StabelVedAggregering<String> stabel = new StabelVedAggregering<>();

    // (3) Sett alle kommandolinjeargumenter (strenger)
    //     inn i stabelen.
    for (int i = 0; i < args.length; i++)
      stabel.puttPå(args[i]);

    // (4) Putt på et heltall for moro skyld.
    // stabel.puttPå(new Integer(100));   // Kompileringsfeil
```

```
    // (5) Hent et element om gangen fra stabelen.
    while (!stabel.tom())
      System.out.print(stabel.taAv().toLowerCase() + " ");
    System.out.println();
  }
}
```

Kjøring av programmet:

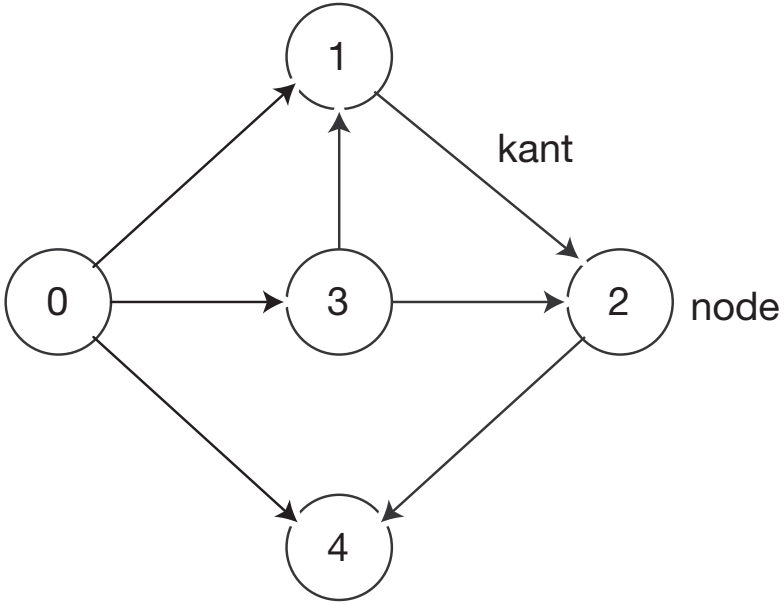
```
>javac StabelKlient.java
>java StabelKlient Dette er kult og moro
moro og kult er dette
```

Bruk av stabler

- Vi skal se på en problemstilling fra *grafteori*.
- En graf har *noder* og *kanter* som binder nodene sammen.
  - Kantene er *rettet*, det vil si at de har et pilhode i den ene enden.
  - Man kan navigere fra en node til en annen i angitt retningen.

Rutenett som graf

- Hver node er en by (som har et nummer angitt i noden)
- En kant fra en by angir hvilken by som kan nås direkte fra denne byen.



(a) Graf

	[0]	[1]	[2]	[3]	[4]
[0]	F	T	F	T	T
[1]	F	F	T	F	F
[2]	F	F	F	F	T
[3]	F	T	T	F	F
[4]	F	F	F	F	F

T:true F:false

(b) Graftabell

Problemstilling

- Hvilke byer kan vi nå fra en gitt by?
  - Vi kan f.eks. nå byene med numrene 1, 2 og 4 med utgangspunkt i bynummer 3.
  - Vi finner stegvis hvilke byer som kan nås direkte fra en by, og gjentar samme prosess for disse byene.
- Prosessen bruker en stabel *og* en mengde.
  - Stabelen holder rede på hvilke byer vi ennå ikke har kontrollert for å komme videre til andre byer.
  - Mengden holder rede på byer som er nådd hittil i prosessen.
  - Hvert steg i prosessen går ut på følgende:

```
Ta av byen som er på toppen av stabelen.
Hvis denne byen ikke er inne i mengden:
    Innsett denne byen i mengden.
    Legg alle byer som kan nås direkte fra denne byen, på stabelen.
```

Steg for å finne byer som kan nås fra bynummer 3

Stegnr.	*Stabel*	*Mengde*
1.	<3>	[]
2.	<1, 2>	[3]
3.	<1, 4>	[3, 2]
4.	<1>	[3, 2, 4]
5.	<2>	[3, 2, 4, 1]
6.	<>	[3, 2, 4, 1]

Eksempel: Graf

```
import java.util.*;
public class StabelKlient2 {
  public static void main(String[] args) {
    // (1) Opprett bygraf.
    boolean[][] byGraf = {
        {false, true,  false, true,  true},
        {false, false, true,  false, false},
        {false, false, false, false, true},
        {false, true,  true,  false, false},
        {false, false, false, false, false}
    };

    // (2) Les bynummer fra terminalen.
    System.out.print("Oppgi startby nr. [0-" + (byGraf.length-1) + "]: ");
    int startByNr = Terminal.lesInt();
    if (startByNr < 0 || startByNr >= byGraf.length) {
      System.out.print("Feil bynr.: " + startByNr);
      return;
    }
```

```
// (3) Opprett en bystabel.
StabelVedAggregering<Integer> byStabel = new StabelVedAggregering<>();

// (4) Opprett en bymengde.
Set<Integer> byMengde = new HashSet<>();

// (5) Legg startby på stabelen.
byStabel.puttPå(startByNr);

// (6) Behandle hver by som finnes på stabelen.
while (!byStabel.tom()) {
  // Ta av inneværende by fra stabelen.
  int inneværendeBynr = byStabel.taAv();
  // Kontroller om denne byen allerede er behandlet.
  if (!byMengde.contains(inneværendeBynr)) {
    // Legg inneværende by til bymengden.
    byMengde.add(inneværendeBynr);
```

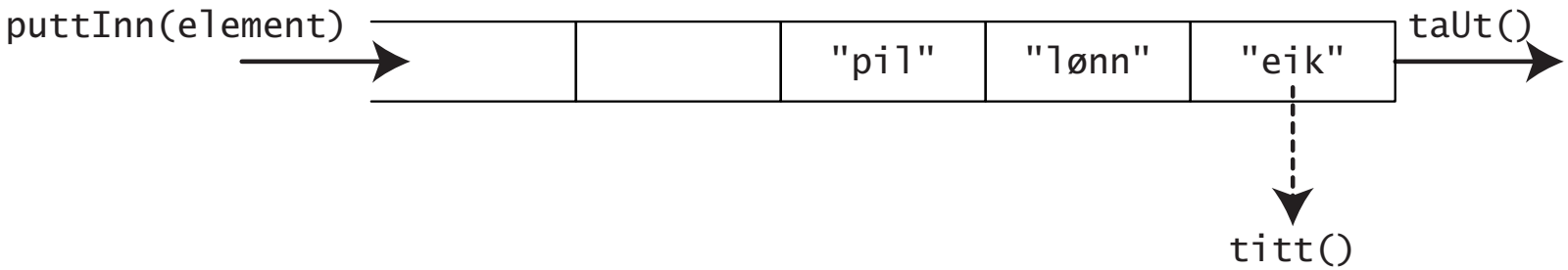
```
        // Legg alle byer som kan nås fra den
        // inneværende byen på bystabelen.
        for (int j = 0; j < byGraf[inneværendeBynr].length; j++) {
          if (byGraf[inneværendeBynr][j])
            byStabel.puttPå(j);
        }
      }
    }
    // Fjern startbyen fra bymengden, og skriv resultat.
    byMengde.remove(startByNr);
    System.out.print("Fra bynummer " + startByNr +
                     " kan følgende by nås: " + byMengde);
  }
}
```

Utføring av program:

```
Oppgi startby nr. [0-4]: 3
Fra bynummer 3 kan følgende by nås: [2, 4, 1]
```

Datastrukturen kø

- En *kø* en *FIFU* (*først-inn, først-ut*)-datastruktur (Figur 9.15).



Køoperasjoner

Køoperasjon	Beskrivelse
`puttInn(element)`	Setter `element` bak i køen (eng. *enqueue()*).
`taUt()`	Fjerner og returnerer det første elementet i køen (eng. *dequeue()*).
`titt()`	Returnerer det første elementet i køen uten å fjerne det fra køen (eng. *peek()*).
`tom()`	Returnerer sann eller usann avhengig av om køen er tom eller ikke (eng. *empty()*).

Køimplementering ved arv

- Siden vi har brukt arv, kan en klient bryte abstraksjonen en kø representerer.

```
class KoeVedArv<E> extends KjedetListe<E> {

  public void puttInn(E data) { innsettBak(data); }    // (1)
  public E taUt() {                                    // (2)
    if (tom()) return null;
    else return slettForan();
  }
  public E titt() {                                    // (3)
    if (tom()) return null;
    return første().hentData();
  }
  public boolean tom() { return erTom(); }             // (4)
}
```

Klient som bruker en kø (KoeKlient.java)

```
public class KoeKlient {

  public static void main(String[] args) {
    // (1) Kontroller om det er noen kommandolinjeargumenter.
    if (args.length == 0) {
      System.out.println("Bruk: java KoeKlient <argumentliste>");
      return;
    }
    // (2) Opprett en kø.
    KoeVedArv<String> kø = new KoeVedArv<>();

    // (3) Sett alle kommandolinjeargumenter (strenger) inn i køen.
    for (int i = 0; i < args.length; i++)
      kø.puttInn(args[i]);

    // (4) Bryter kø-regler.
    kø.innsettForan("snik");
```

```
    // (5) Hent ett element om gangen fra køen.
    while (!kø.tom())
      System.out.print(kø.taUt().toUpperCase() + " ");
    System.out.println();
  }
}
```

Kjøring av program:

```
>javac KoeKlient.java
>java KoeKlient Dette er kult og moro
SNIK DETTE ER KULT OG MORO
```