Lenkelister, iteratorer, indre klasser

Repetisjonskurs våren 2019
kristijb

Lenket liste av objekter

Vi lager en “lenke” ved at objekter refererer til hverandre.

Vanlige er ofte å ha Node-objekter som har en referanse til en annen Node (feks. neste), samt en referanse til et data-Objekt vi ønsker å lagre i beholderen vår/den lenkede listen.

Har gjerne referanse til foran/første Node i Lenkeliste-klassen.

Hvorfor lage egne beholdere?

En lenkeliste er en svært enkel datastruktur å implementere.

Selv om Javas standardbibliotek har en lenkeliste-implementasjon *LinkedList* , så kan det være nyttig å lage egne implementasjoner når vi har spesielle behov.

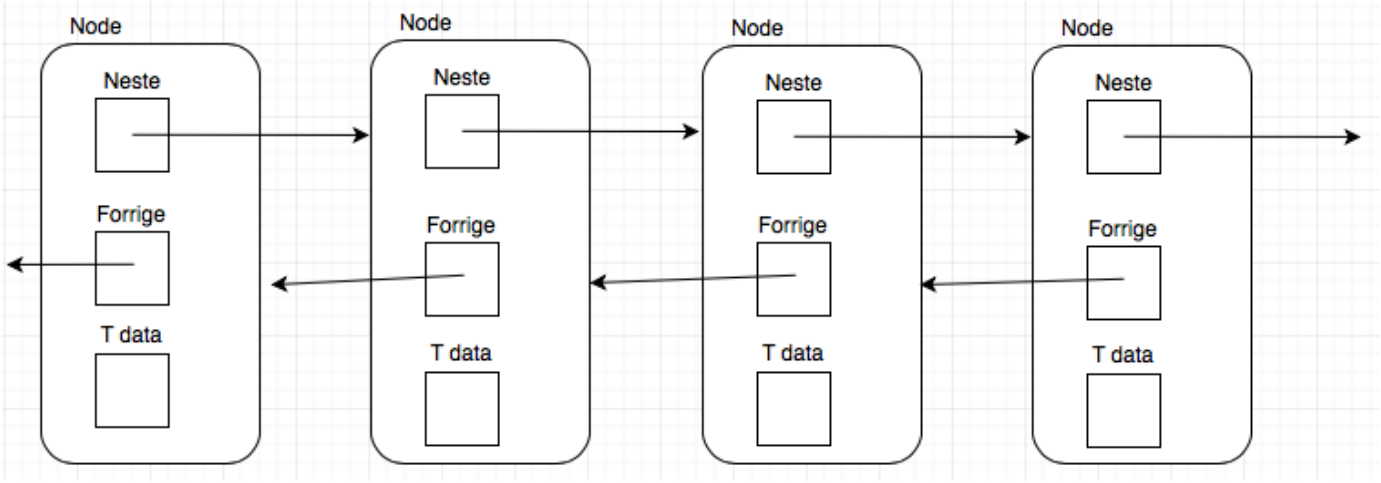
Noen programmeringsspråk (som f.eks. C) har ikke noen implementasjon i sitt standardbibliotek, så da blir en nødt til å lage alt fra bunnen av.

Dobbellenket

Dobbel: både referanse til forrige og neste.

Ulempe: flere referanser å holde styr på.

Fordel: kan enkelt gå begge veier i listen.

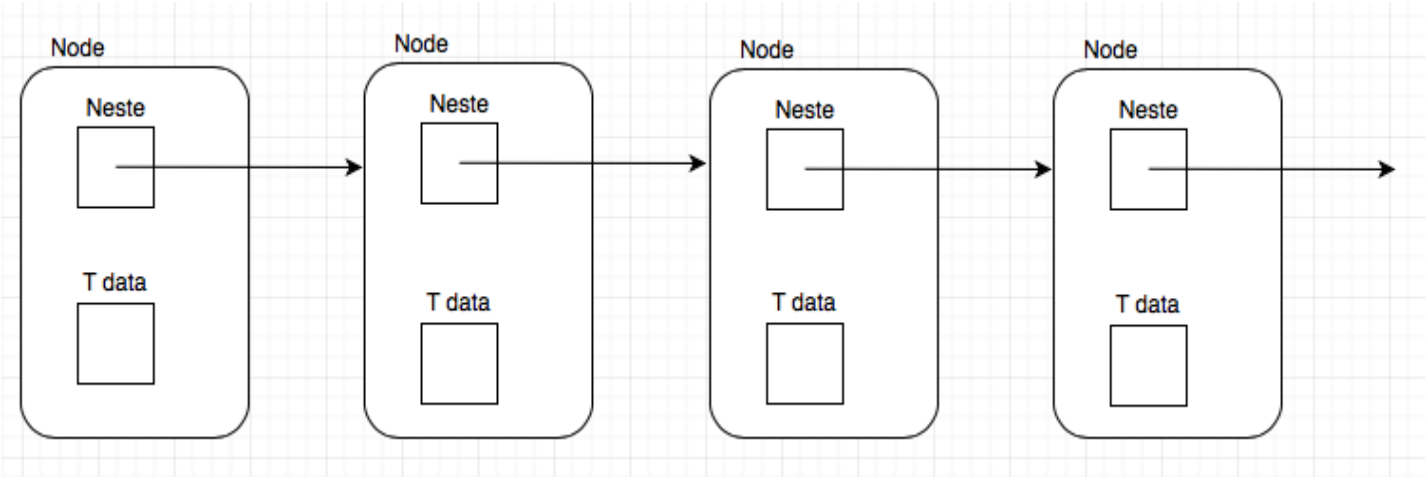


Enkeltlenket

Enkel: kun referanse til neste.

Ulempe: kan kun gå fra foran/første node og bakover.

Fordel: færre referanser å holde styr på, og derfor kanskje enklere å få satt alt riktig.



Node-klassen

To alternativer:

LenkeListe har en (privat) *indre* klasse Node

Hvis listen skal være generisk får vi: LenkeListe<T> og Node

Fordel: Slipper å la Node være generisk siden T er *bundet* inne i LenkeListe<T>

Ulempe: Hver implementasjon må ha sin egen Node-klasse.

LenkeListe og Node er vanlige klasser

Hvis listen skal være generisk får vi: LenkeListe<T> og Node<T>

Fordel: Mulig å gjenbruke Node<T>

Ulempe: Litt ekstra kode fordi typen til nodene i lenkelisten må være Node<T>.

Eksempel-kode: generisk klasse Lenkelise, privat indre klasse Node

```
class Lenkeliste <T> {
    Node foran;

    private class Node {
        T data;
        Node neste;

        public Node(T data){
            this.data = data;
        }
    }
```

Indre klasser

Når ingen andre enn én klasse har behov for et type objekt, kan objektets klasse lagres som en *indre* klasse.

For Lenkelister er det (ofte) kun lenkelisten selv som har behov for Noder (utenfor er man kun interessert i dataen som Nodene holder).

Iteratorer er ofte indre klasser fordi de er definert til å fungere på en spesifikk liste (iteratorens implementasjon er tilpasset listens implementasjon), ingen andre skal bruke iteratoren.

Anonyme klasser er en type indre klasse.

Innsetting og fjerning

LIFO, FIFO, Ordnet

FIFO - first in, first out. Kø.

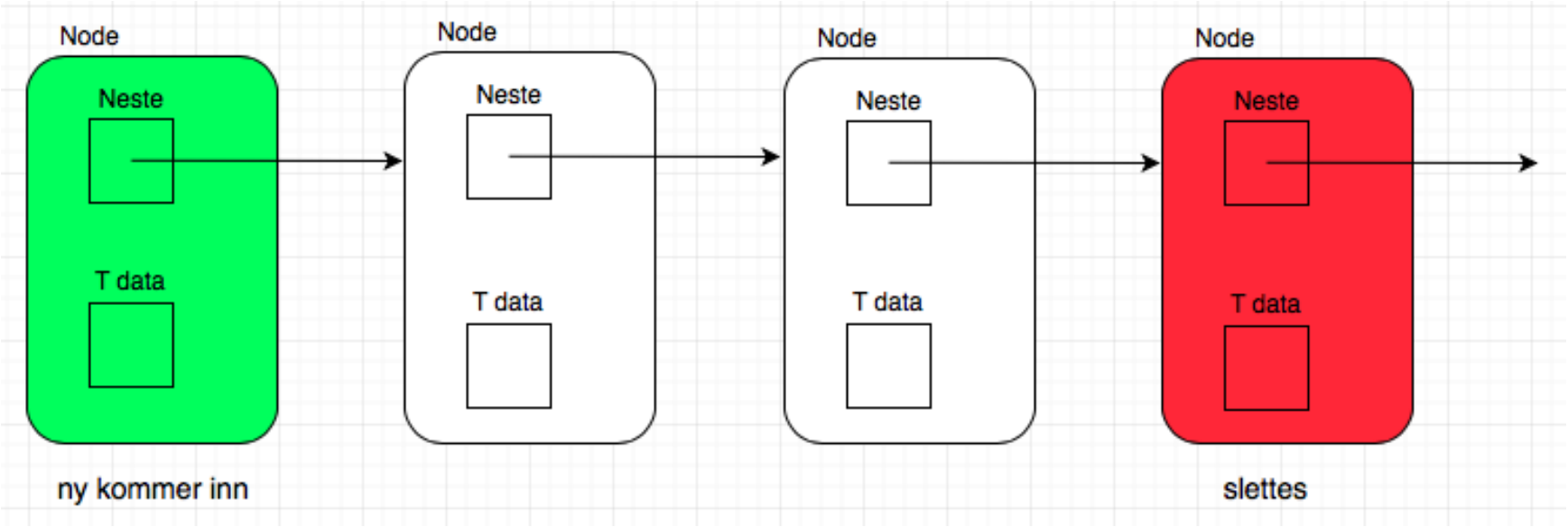
LIFO - last in, first out. Stack.

Ordnet - i sortert rekkefølge. Klassen

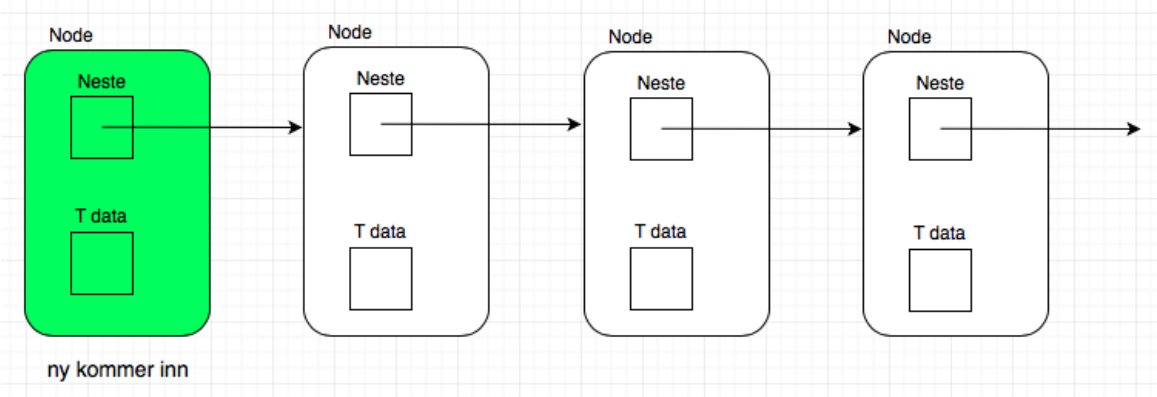
det skal sorteres på må da ha

compareTo-grensesnittet implementert.

Innsetting og fjerning: FIFO



Innsetting og fjerning: LIFO



Insetting, foran og bak i en enkel lenkeliste

```
void settInnForan(T data) {
    Node ny = new Node(data);
    ny.neste = foran;
    foran = ny;
}

void settInnBak(T data) {
    Node ny = new Node(data);
    Node temp = foran;
    if(foran == null) {
        foran = ny;
        return;
    }
    while(temp.neste != null) {
        temp = temp.neste;
    }
    temp.neste = ny;
}
```

Fjerning, foran og bak i en enkel lenkeliste

```
boolean taUtForan() {
    if (foran == null) {
        return false;
    }
    foran = foran.neste;
    return true;

}

boolean taUtBak() {
    if (foran == null) {
        return false;
    }
    if (foran.neste == null) {
        foran = null;
        return true;
    }
    Node temp = foran;
    //Dersom temp.neste.neste er null saa skal
    //temp.neste det siste elementet i listen
    //og den som skal ut.
    while(temp.neste.neste != null) {
        temp = temp.neste;
    }
    temp.neste = null;
    return true;
```

Iterator og Iteratble-grensesnittene

Iterator - grensesnittet som implementeres på (den indre) klassen iterator (kall den hva du vil feks MinIterator: “class MinIterator implements Iterator “

Iterable - grensesnittet som implementers på listen/beholderen som skal bli itererbar.

Fortsetter med samme enkle lenkeliste-eksemepel

```
import java.util.Iterator;

class Lenkeliste <T> implements Iterable<T> {
    Node foran;

    Iterator<T> iterator(){}

    private class LenkelisteIterator implements Iterator<T> {
        T next(){}
        boolean hasNext(){}
    }
```

Iterable

public Iterator<T> iterator()

> Returnerer en ny iterator over listen.

Iterator

boolean hasNext() - returnerer true dersom det finnes et neste element og false dersom det ikke gjør det.

T next() - returnerer innholdet/data i neste Node.

Implementert for vårt eksempel:

I Lenkeliste-klassen:

```
class Lenkeliste <T> implements Iterable<T> {
    Node foran;

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

Implementert for vårt eksempel:

I LenkelisteIterator-klassen:

```
private class LenkelisteIterator implements Iterator<T> {
    //må holde på posisjonen vår. Starter foran.
    Node posisjon;
    public LenkelisteIterator(){
        posisjon = foran;
    }

    T next() {
        T returdata = posisjon.data;
        posisjon  = posisjon.neste;
        return returdata;

    }

    boolean hasNext() {
        return posisjon != null;
        // hvis posisjon == null finnes det ikke noe mer i listen
        // hvis posisjon != null finnes det mer i listen
    }
}
```

Annet

Listehode og listehale - tomme noder som alltid er der, lagrer ikke data.

Fordel: slipper å sjekke for en del spesialtilfeller. Når listen er tom

er listehode == listehale.

bakerst-peker i tillegg til foran - praktisk når man skal sette nye elementer inn bakerst, eller for en dobbeltlenket liste (kan da enkelt gå begge veier, begynne fra bakerst).