# Beholdere og generiske klasser II

IN1010 uke 7

Tirsdag 7. mars 2023

Institutt for informatikk, Universitetet i Oslo

# Beholdere og generiske klasser - II

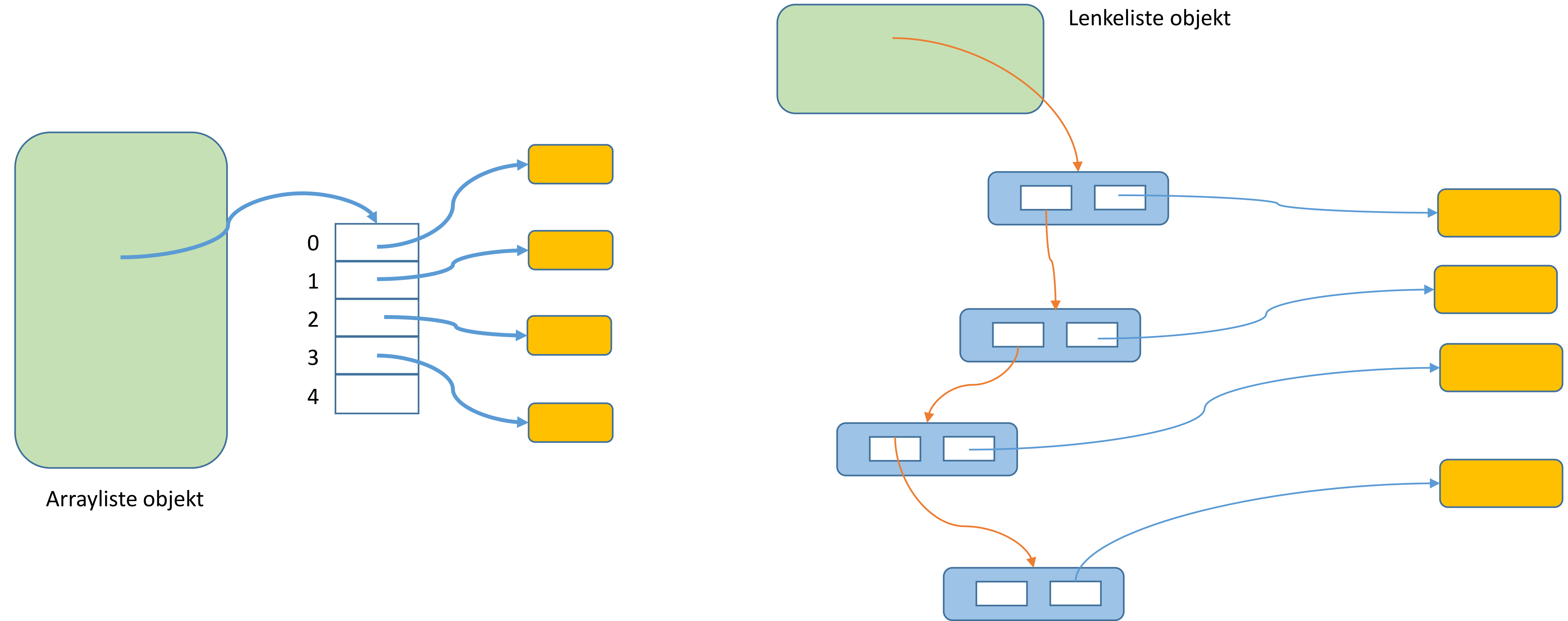
- Implementasjon av lenkelister (ulik datarepresentasjon i beholderne)
    - Enveislister (som vi så på sist)
    - Toveislister (nytt)
    - Egen referanse til siste element i listen

- Varianter av lister (ulik semantikk)
    - stabel (stack, Last In First Out – LIFO)
    - kø (First In First Out – FIFO)
    - Prioritetskø

- Mer Java
    - Innpakking ("boxing") av primitive typer
    - Å sammenligne objekter (**Comparable** )
    - Å gå gjennom alle elementer i en beholder (**Iterator**)

Repetisjon:

# Liste implementert med array versus lenkeliste



Repetisjon:

# Hva gjør egentlig `peker = peker.neste`?

```java
public int size() {
    int teller = 0;
    Node peker = start;
    while (peker != null) {
        teller++;
        peker = peker.neste;
    }
    return teller;
}
```

Variabel som skal få ny verdi

uttrykk som evaluerer til verdi

`peker = peker.neste`

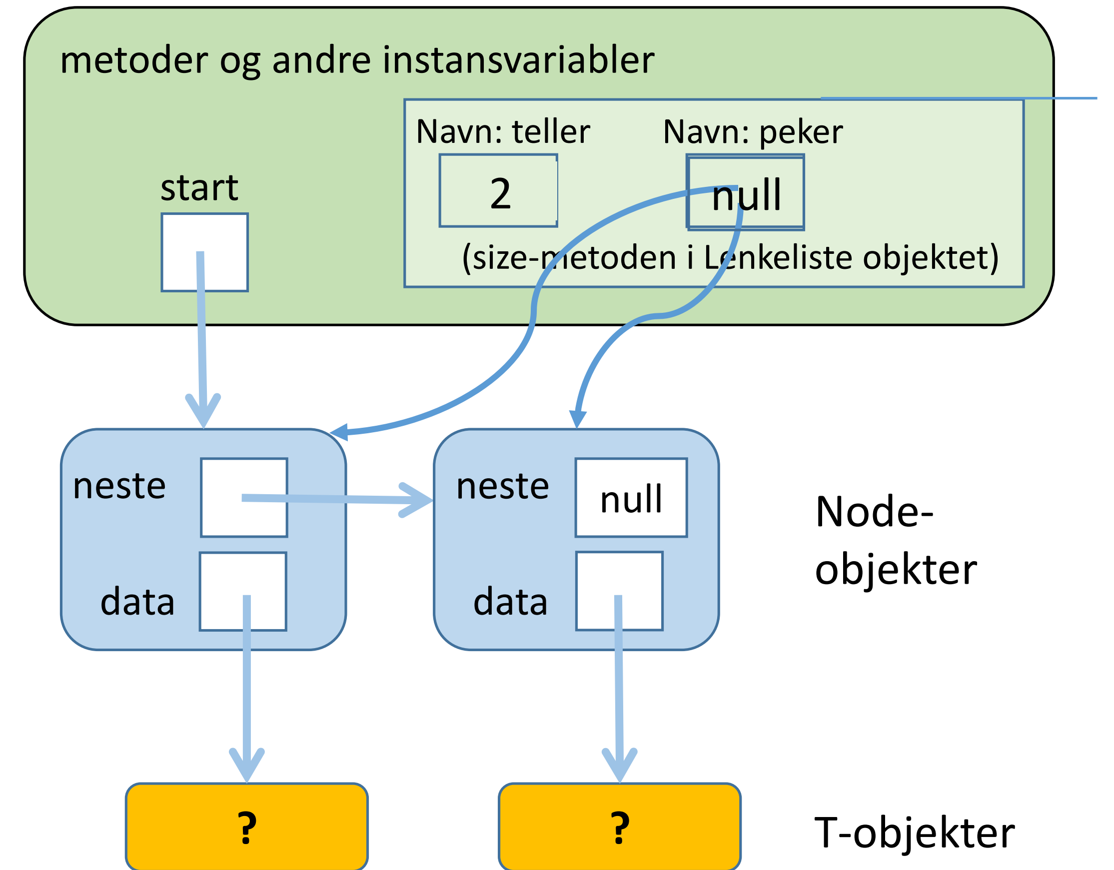
Verdi: innholdet i instansvariabelen **neste** i objektet **peker** refererer til

- På venstresiden av en tilordning angir et variabelnavn en **minnelokasjon** som skal få en ny verdi
- Alle andre steder vi bruker et variabelnavn angir det **verdien** som ligger i variabelen nå
  - Høyresiden i en tilordning
  - Som argument til et metodekall
  - Som del av et sammensatt uttrykk, f eks **peker.neste** eller **x+2** eller **s+"\n"**

Repetisjon:

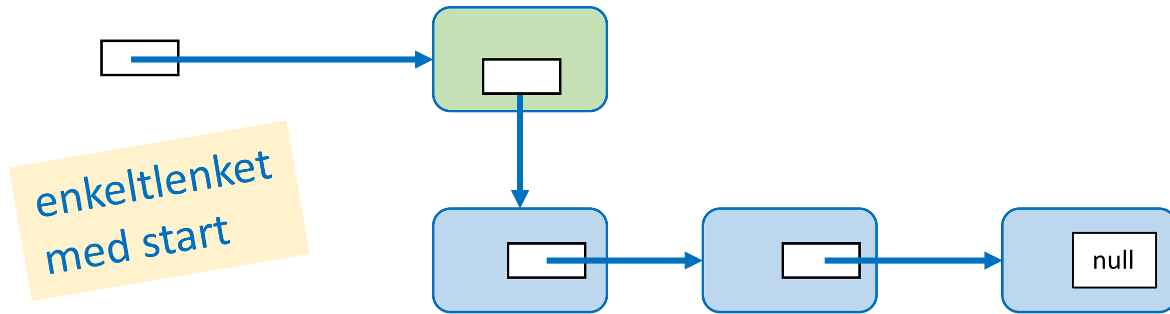
# Hva gjør egentlig **`peker = peker.neste`**?

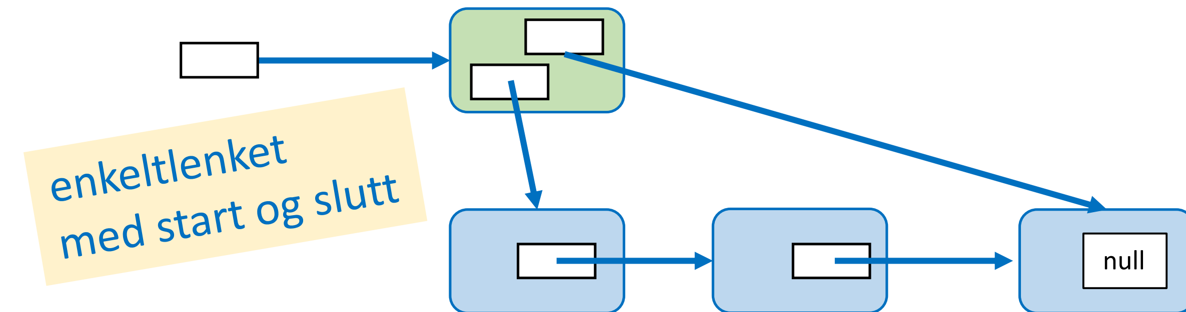
```
public int size() {
    int teller = 0;
    Node peker = start;
    while (peker != null) {
        teller++;
        peker = peker.neste;
    }
    return teller;
}
```

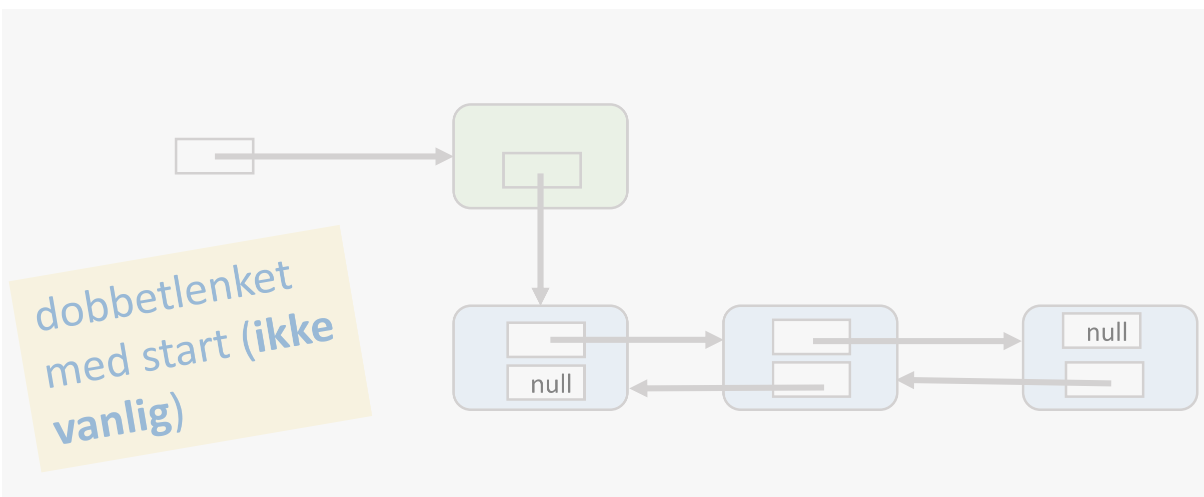


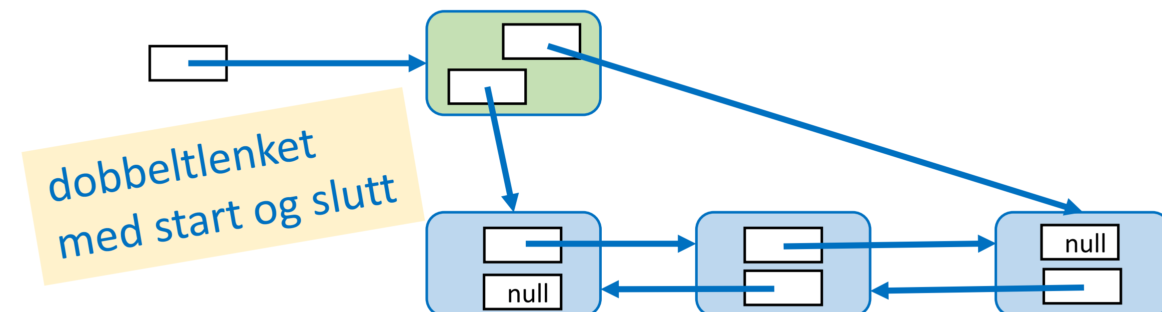
# Lenkelister: Alternative datastrukturer

## + med eller uten instansvariabel **size**

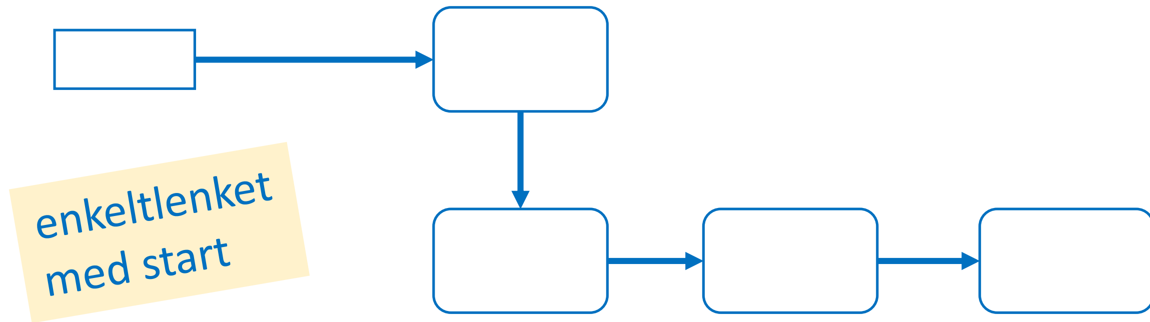


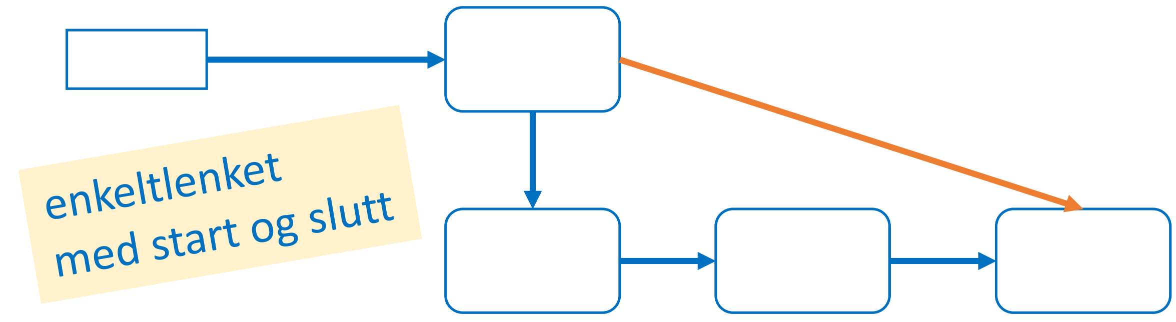



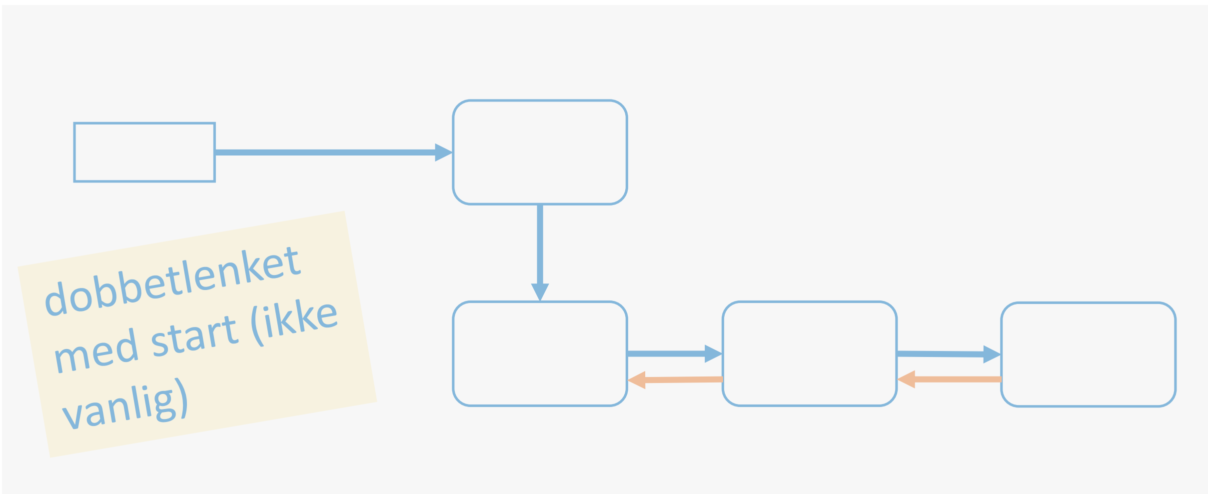



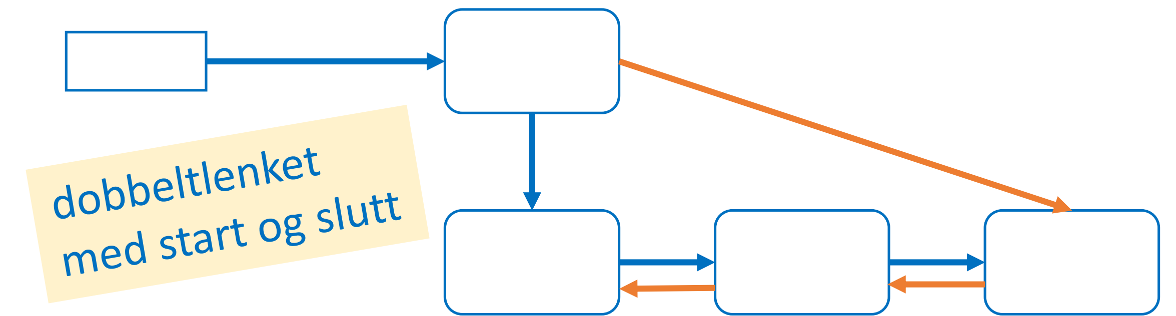



# Lenkelister – alternative datastrukturer

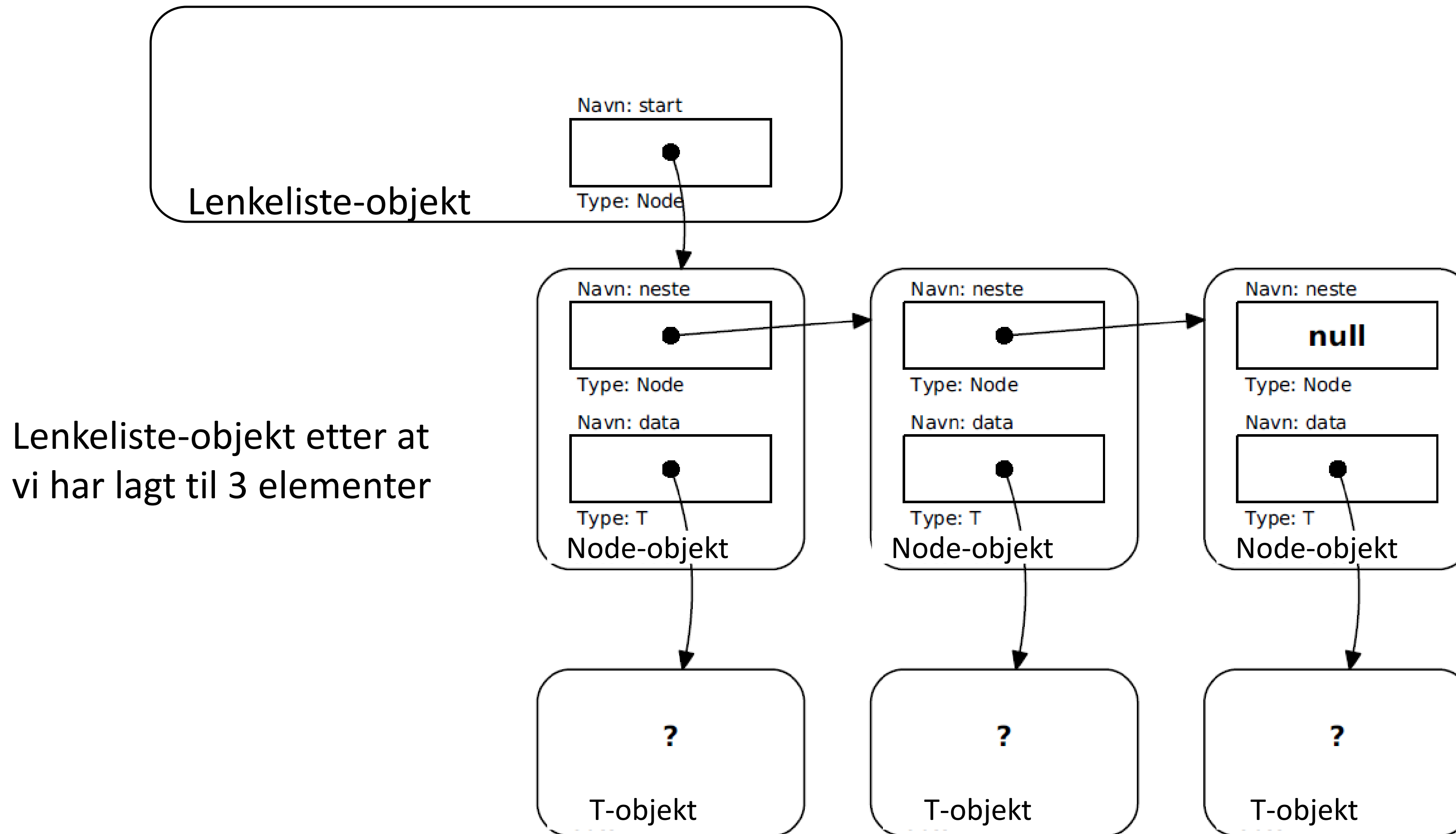


# Forrige uke: Klassen Lenkeliste




Lenkeliste-objekt etter at vi har lagt til 3 elementer

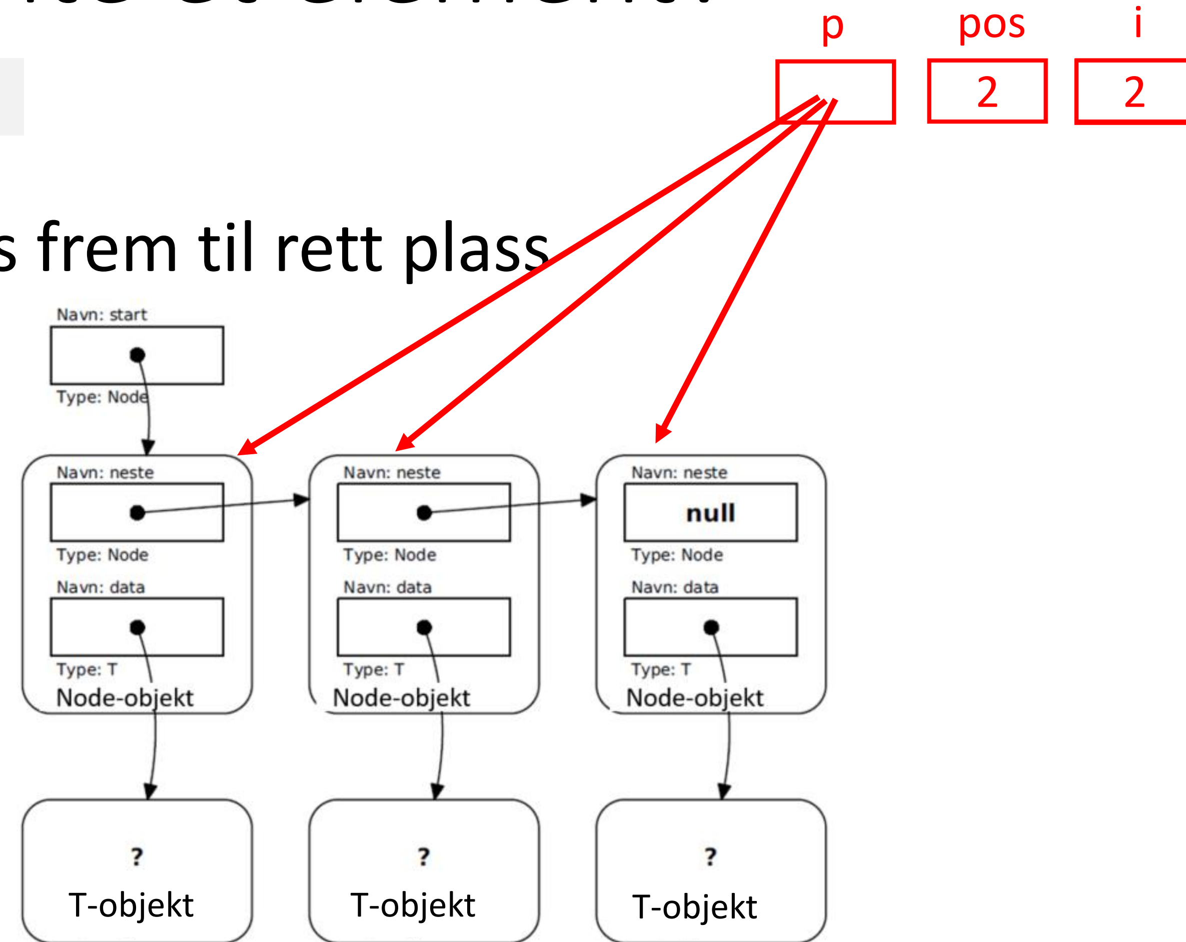
# Forrige uke: Hvordan hente et element?

```
public T get(int pos) {}
```

- Går gjennom liste, teller oss frem til rett plass

```
Node p = start;
for (int i=0;i<pos;i++) {
    p = p.neste;
}
```

- NB: Hva skal vi returnere?



# Fordeler og ulemper ved enkeltlenket liste

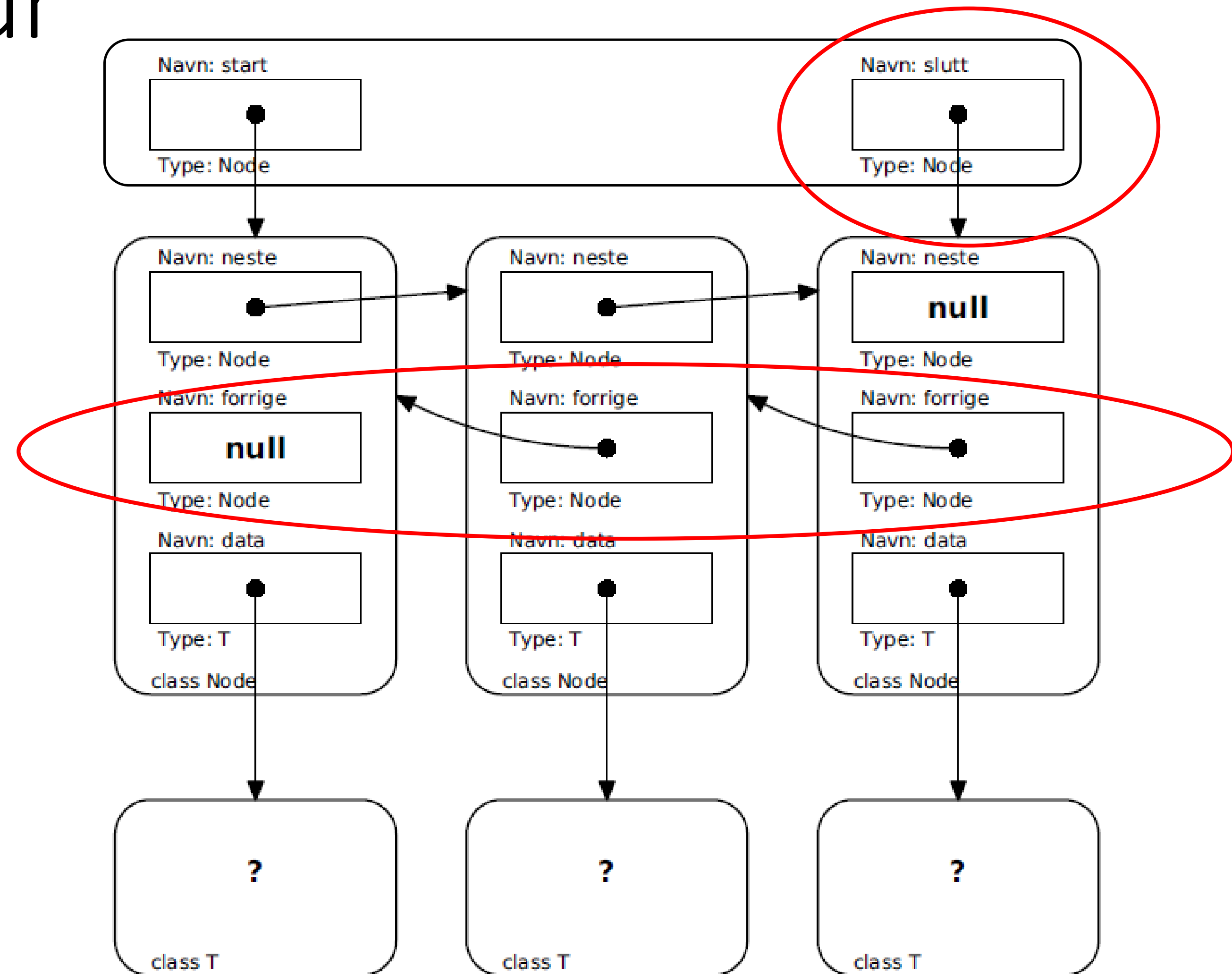
Både effektivitet ved kjøring, og effektivitet (og kvalitet) i utvikling, kan være relevante hensyn

+ Fleksibel! Alltid plass til akkurat så mange elementer som trengs
+ (relativt) enkelt å programmere metodene

- Mye "nesting" for å jobbe seg frem til elementer langt ut i listen
- Må passe på å stoppe én node foran den som skal fjernes

# Toveisliste datastruktur

```
class Node {
      Node neste = null;
      Node forrige = null;
      T data;
      Node(T x) {
            data = x;
      }
}
private Node start=null;
private Node slutt=null;
```

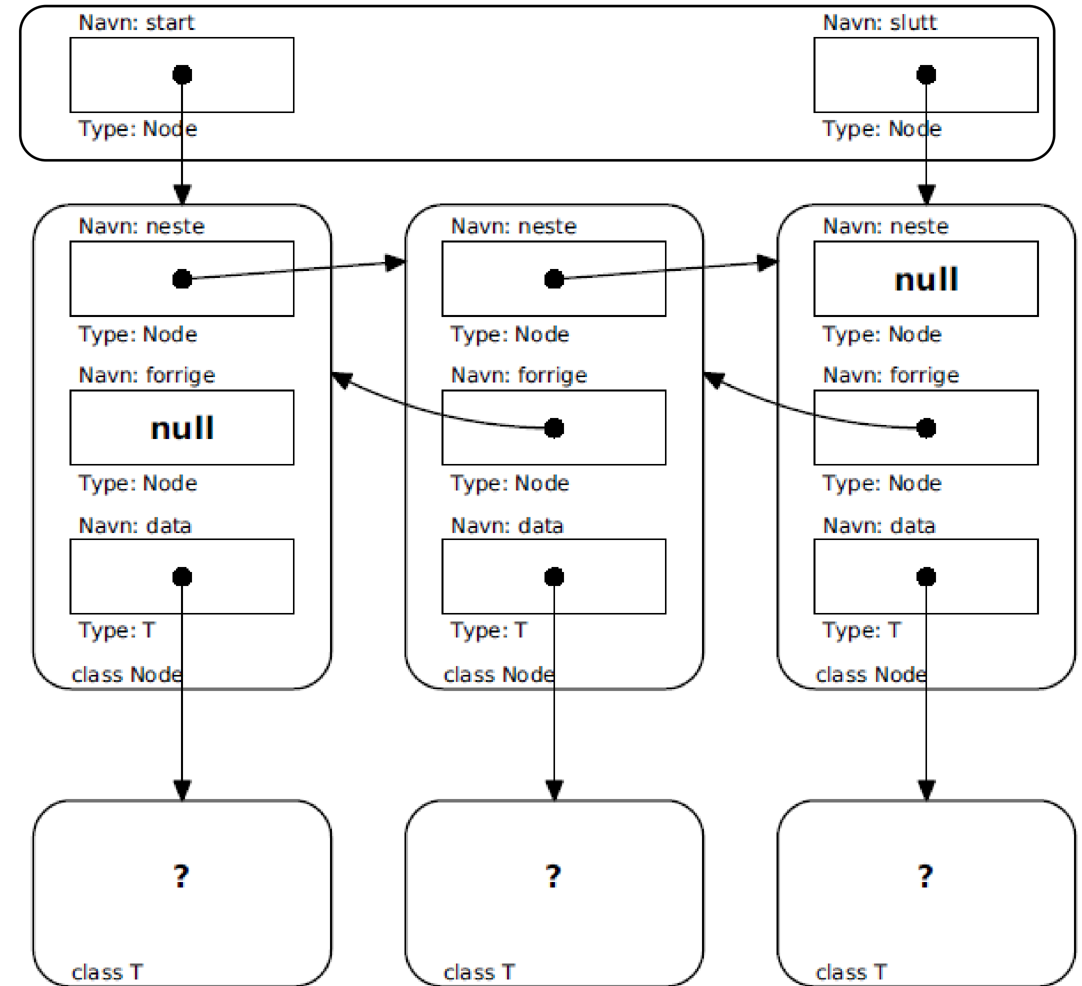


# Toveisliste

- Slipper å lete oss frem til siste element ved innsetting, fjerning, endring
- Eks - fjerne siste element:

```
Node n = slutt;
slutt.forrige.neste = null;
slutt = slutt.forrige;
```

- Støtter baklengs traversering av listen



# Valg av datastruktur for en liste

- Det er vanligvis fornuftig å ha en instansvariabel **slutt** også i en enveis liste
  - Enkelt å lese av/ endre det siste, og legge til nytt element bakerst

- Det kan være fornuftig å velge toveis liste om
  - man man trenger å traversere listen i begge retninger, eller
  - man ofte skal fjerne andre elementer enn det første

- Ellers greit å bruke enveis liste - strukturen og (stort sett) programmeringen blir enklere

- En instansvariabel for størrelse tar liten plass og gjør at man ikke trenger traversere hver gang for å finne størrelsen

# Dataelementer av primitive typer

- Klasseparametere kan kun representere referanse-typer!
- Hvordan kan vi lagre og organisere for eksempel heltall eller boolske verdier?
- Pakker den enkle verdien inn i et objekt av passende klasse: Integer, Boolean => *Boxing*
- Java hjelper oss med automatisk inn- og utpakking for standardklassene:

```
ArrayList<Integer> lx = new ArrayList<>();
lx.add(12);
int v = lx.get(0);
```

# Noen klassiske hovedtyper beholdere

Sist implementerte vi en Arrayliste med operasjoner på bestemte posisjoner (og **add**, som alltid la til sist i Lenkeliste)

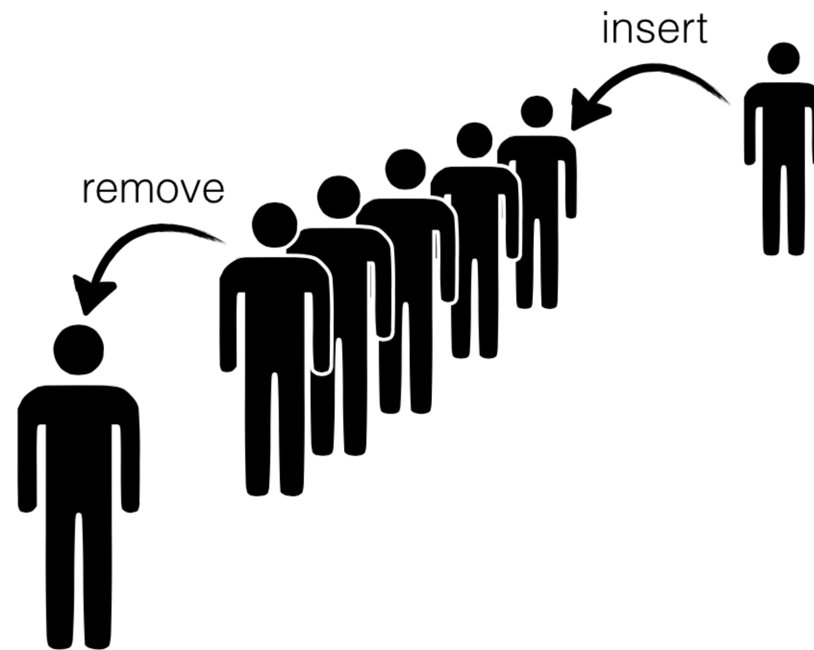
Hva avgjør hvilket element som hentes ut?

- Kø
- Stabel
- Prioritetskø

Vi kan lage subklasser av Arrayliste der get(0) gir ulikt resultat avhengig av om subklassen implementerer en kø, stabel eller prioritetskø.

# Kø («queue»)
# First in First Out - FIFO

- En type lister der vi alltid setter inn nye elementer bakerst og henter dem ut fra starten.



# Inn i / ut av en kø

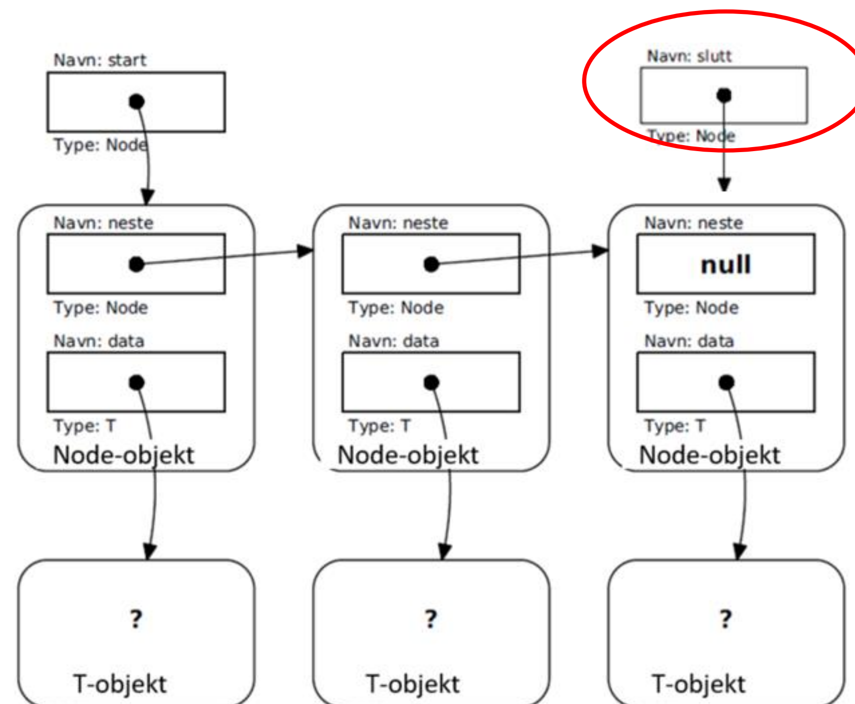
- Sette inn i kø (bakerst):

```
Node ny = new Node(v);
slutt.neste = ny;
slutt = ny;
```

(Men hva om køen er tom når vi skal sette inn?)

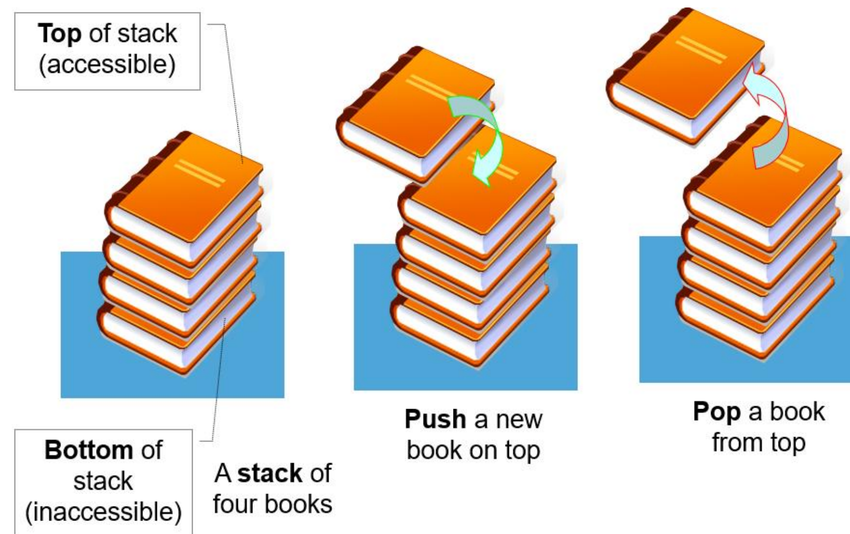
- Hente ut av køen (første element)

```
T svar = null;
if (start != null) {
    svar = start.data;
    start = start.neste;
}
```



# Stabel («stack»)
# Last in First Out – LIFO

- *Stabler* er lister der vi legger nye elementer på toppen («push-er») og henter fra samme ende («pop-er»).
- Vi jobber hele tiden på toppen = i begynnelsen av listen

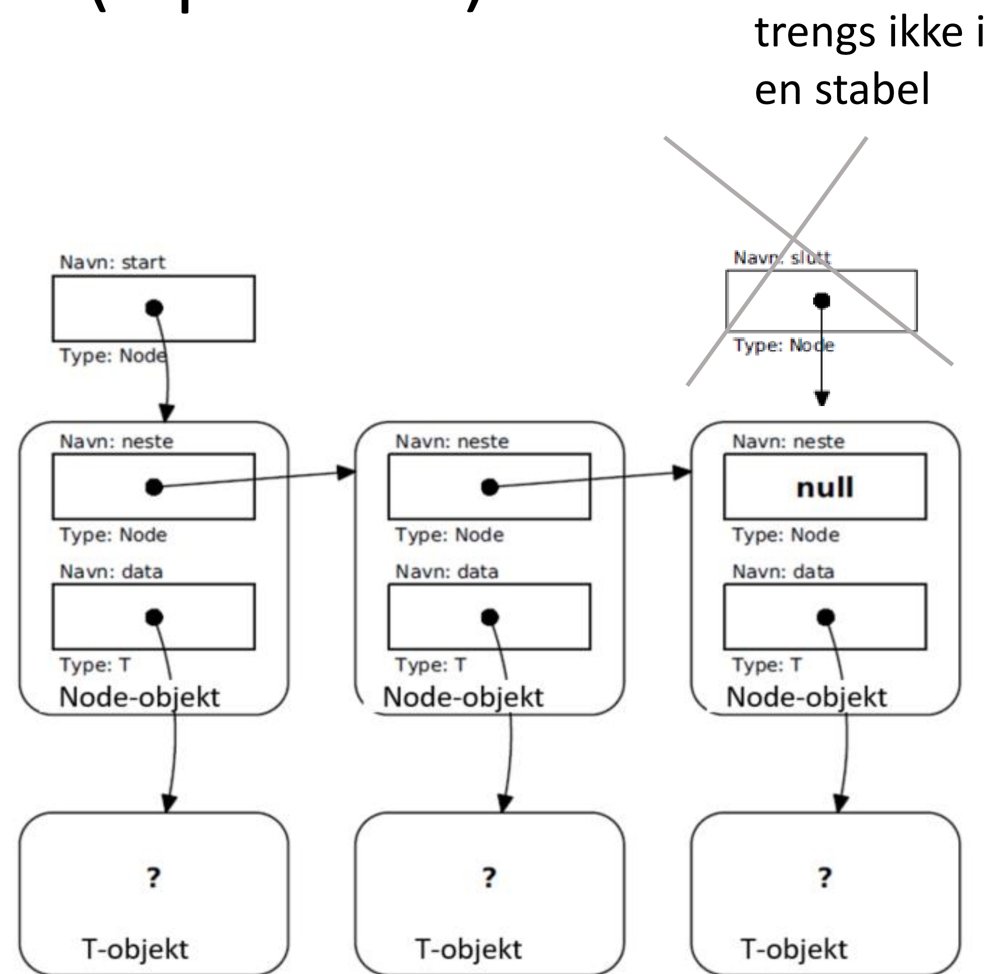


# Legge til nytt element («push»)

trengs ikke i en stabel

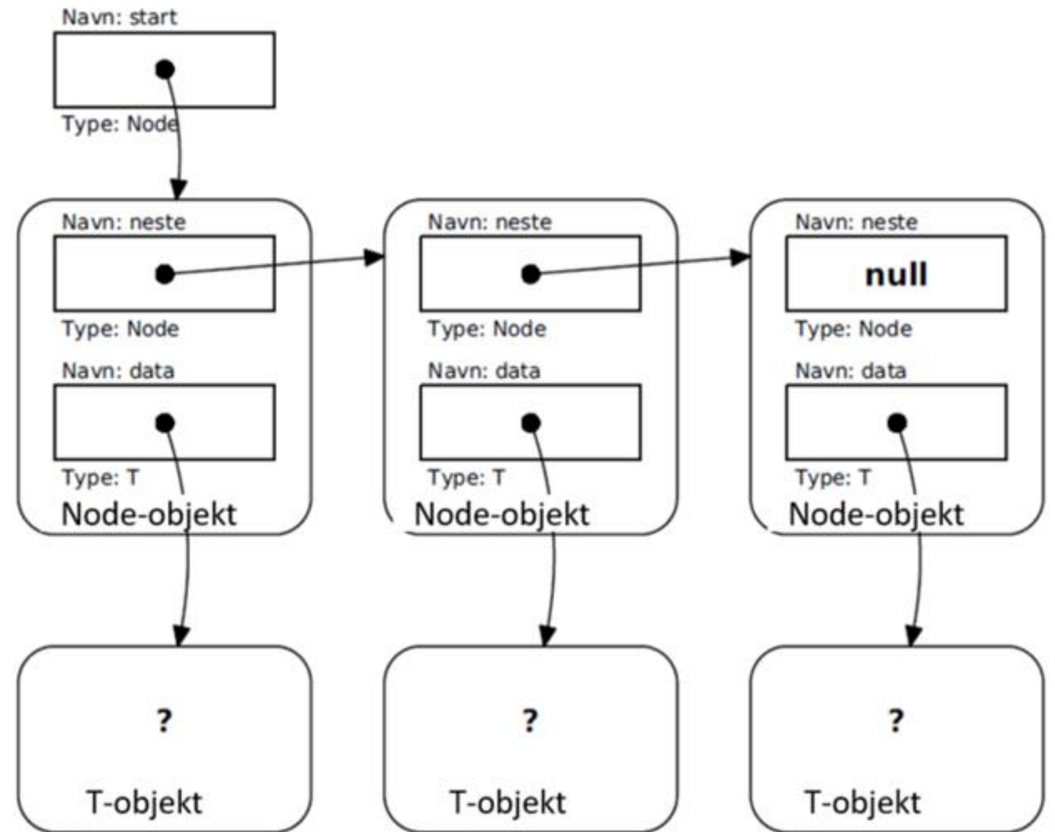
```
Node ny = new Node(v);
ny.neste = start;
start = ny;
```

(Fungerer bra også om stabelen er tom.)



# Ta av element («pop»)

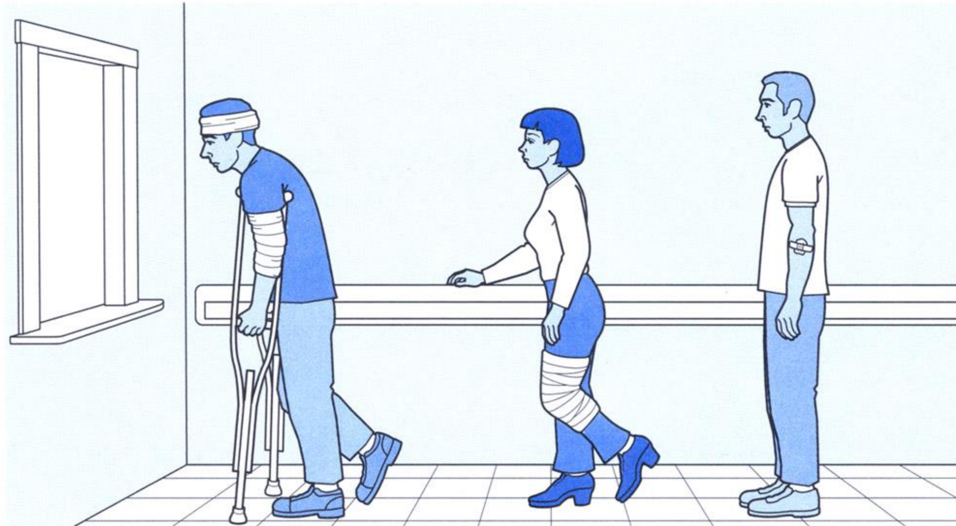
```
T svar = null;
if (start != null) {
    svar = start.data;
    start = start.neste;
}
```



# Prioritetskø («Priority queue»)

- I prioritetskøer haster noen elementer mer enn andre; de tas først uavhengig av hvor lenge de har stått i køen.
- Høyeste prioritet = laveste verdi skal prioriteres

NB



# Implementere en prioritetskø

Invariant for prioritetskøen: Et element er alltid nærmere starten på listen enn alle elementer med lavere prioritet

- Vil holde vår prioritetskø sortert ved at *nye elementer* settes på korrekt plass: Den første i køen skal være først i listen

- Hvordan vet vi hva som er riktig sortering?

- Vi trenger en standardtest som forteller oss om forholdet for to elementer p og q:
  - Er $p < q$?
  - Er $p = q$?
  - Er $p > q$?

# Interface Comparable

- Java-biblioteket har et interface kalt **Comparable** som angir at noe er sammenlignbart. Det inneholder kun én metode

```
public interface Comparable<T> {
    public int compareTo(T otherObj);
}
```

- "Vårt" objekt kan sammenlignes med et annet objekt og returnere et heltall
  - < 0 hvis objektet vi kaller compareTo på er "mindre" enn det andre
  - = 0 hvis objektet vi kaller compareTo på er "likt" det andre
  - > 0 hvis objektet vi kaller compareTo på er "større" enn det andre

# Eksempel Comparable: Resultatliste OL

- Skal rangere landene i vinter-OL etter antall medaljer
- Antall gullmedaljer teller mest
- .. men ved like mange gull, avgjør antall sølvmedaljer
- .. ved like mange gull og sølv, avgjør antall bronsjemedaljer
- .. om de har like mange medaljer i hver valør er to land "like"

- Må kunne sammenligne to og to land
- Trenger en metode som kan kalles på ett objekt med et annet objekt som parameter
- Sammenligningen avhenger av datastrukturen og semantikken i objektenes klasse
  ... dvs denne metoden må skrives på nytt for hver klasse!
- MEN Java har et interface Comparable som standardiserer måten å gjøre dette på

# interface Comparable for Deltagerland

```
public class Deltagerland implements Comparable<Deltagerland> {
    private String navn;
    private int antGull, antSoelv, antBronse;
    Deltagerland(String id, int g, int s, int b) {
        navn = id; antGull = g; antSoelv = s; antBronse = b; }

    @Override
    public int compareTo(Deltagerland a) {
        if (antGull < a.antGull) return -1;
        if (antGull > a.antGull) return 1;
        if (antSoelv < a.antSoelv) return -1;
        if (antSoelv > a.antSoelv) return 1;
        if (antBronse < a.antBronse) return -1;
        if (antBronse > a.antBronse) return 1;
        return 0;
        }
    // andre metoder
}
```

# Tester sammenligning av deltagerland

```
class TestMedaljer {
    public static void main(String[] args) {
        Deltagerland danmark = new Deltagerland("Danmark", 0, 0, 0),
                     finland = new Deltagerland("Finland", 1, 1, 4),
                     island = new Deltagerland("Island", 0, 0, 0),
                     norge = new Deltagerland("Norge", 14, 14, 11),
                     sverige = new Deltagerland("Sverige", 7, 6, 1);
        System.out.println("Finland vs Sverige: " + finland.compareTo(sverige));
        System.out.println("Norge vs Sverige: " + norge.compareTo(sverige));
        System.out.println("Danmark vs Sverige: " + danmark.compareTo(sverige));
        System.out.println("Danmark vs Island: " + danmark.compareTo(island));
        }
    }
```

```
Finland vs Sverige: -1
Norge vs Sverige: 1
Danmark vs Sverige: -1
Danmark vs Island: 0
```

# Sorterer vha Comparable

```
import java.util.Arrays;
public class Resultatliste {
    public static void main(String[] args) {
        Deltagerland[] land = {
            new Deltagerland("Danmark", 0, 0, 0),
            new Deltagerland("Finland", 2, 2, 4),
            new Deltagerland("Island", 0, 0, 0),
            new Deltagerland("Norge 2022", 16, 8, 13),
            new Deltagerland("Sverige", 8, 5, 5),
            new Deltagerland("Norge 2018", 14, 14, 11)};
        Arrays.sort(land);
        for (int i = 0; i < land.length; i++)
            System.out.println(land[i].hentNavn());
    }
}
```

```
Danmark
Island
Finland
Sverige
Norge 2018
Norge 2022
```

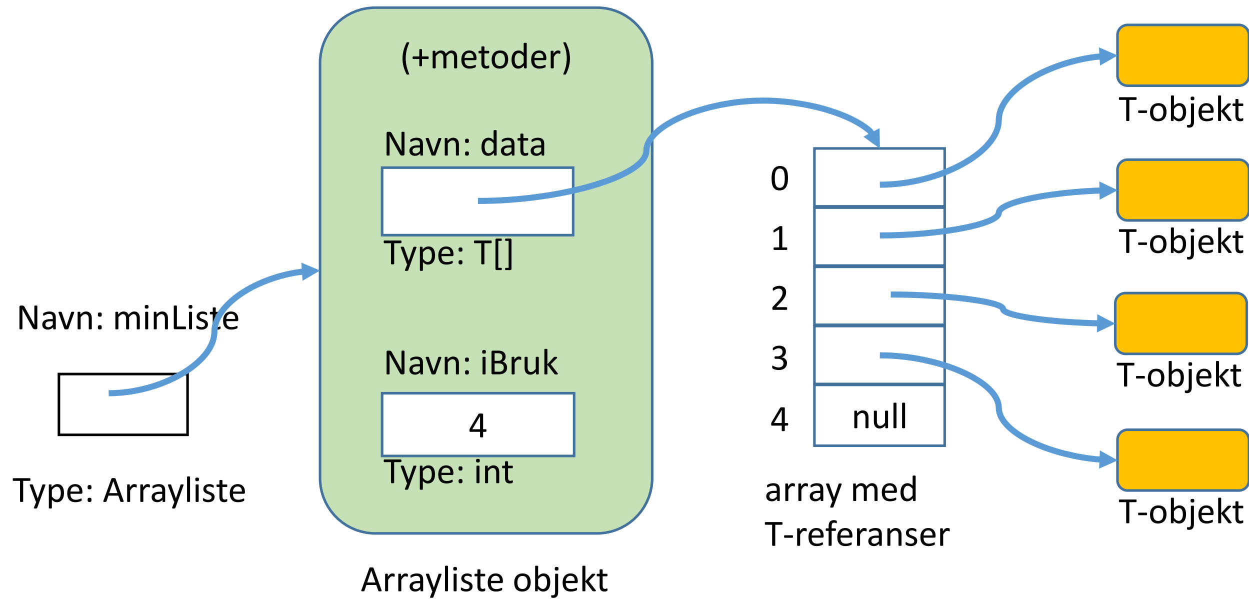
!

# Sorterer vha Comparable

```java
import java.util.Arrays;
public class ResultatlisteReversert {
    public static void main(String[] args) {
        Deltagerland[] land = {
            new Deltagerland("Danmark", 0, 0, 0),
            new Deltagerland("Finland", 2, 2, 4),
            new Deltagerland("Island", 0, 0, 0),
            new Deltagerland("Norge 2022", 16, 8, 13),
            new Deltagerland("Sverige", 8, 5, 5),
            new Deltagerland("Norge 2018", 14, 14, 11)};
        Arrays.sort(land);
        for (int i = land.length-1; i >=0 ; i--)
            System.out.println(land[i].hentNavn());
    }
}
```

```
Norge 2022
Norge 2018
Sverige
Finland
Island
Danmark
```

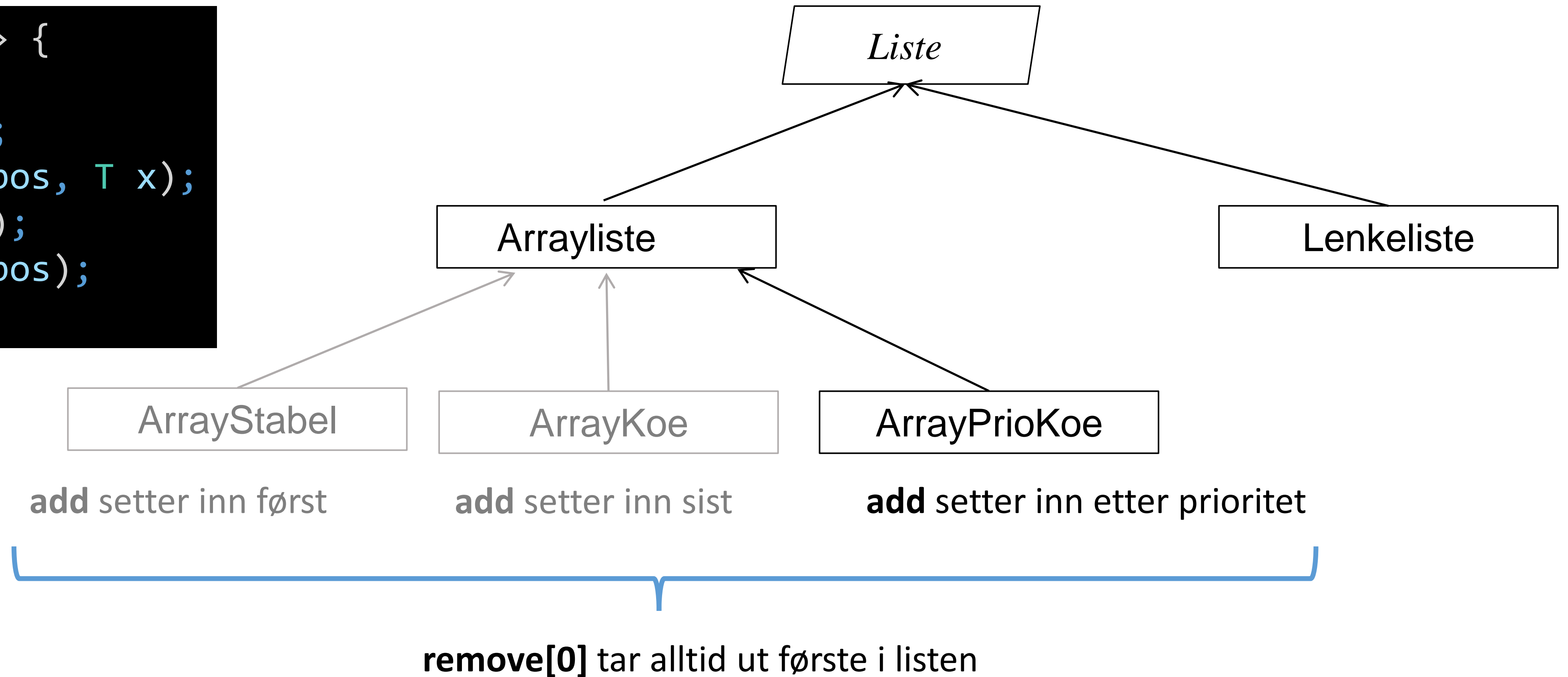
# Implementasjon av prioritetskø




Vi kan lage en prioritetskø som en spesialisering av Arrayliste.

# "Vårt" typehierarki
## (ikke det samme som i Java API eller oblig)

```
interface Liste<T> {
    int size();
    void add(T x);
    void set(int pos, T x);
    T get(int pos);
    T remove(int pos);
}
```



# Implementasjon av prioritetskø

- Vi kan lage ArrayPrioKoe som en spesialisering (subklasse) av Arrayliste.
- I Arrayliste satte vi inn nye elementer bakerst
- ... og så på eller fjernet etter posisjon, med **get** eller **remove**

- Vi bestemmer oss for en minimal endring av Arrayliste:
  - endrer **add** til alltid å sette inn slik at førsteprioritet (lavest verdi!!) er først i listen
  - **remove(0)** vil ta ut elementet først i køen (som før)
  - **get(pos)** vil la oss se hvilket element som er nummer **pos** i køen (som før)

# Klasse som implementerer prioritetskø ved hjelp av Arrayliste

- Klassen ArrayPrioKoe er en subklasse av Arrayliste.

- Klasseparameteren T må implementere Comparable så vi får en sammenligning å sortere etter (og denne sammenligningen må sammenfalle med prioritet)

- Vi redefinerer kun metoden add siden innsettingen skal gjøres sortert.
  De øvrige metodene i Arrayliste kan være som de er.

```
public class ArrayPrioKoe<T extends Comparable<T>> extends Arrayliste<T> {
    @Override
    public void add(T x) {
```

# **add**-metode for prioritetskø

Identifiserer spesialtilfeller:

- Listen kan være tom fra før av.

- Hvis ikke, leter vi oss frem til første element i listen som er større enn det vi skal sette inn.
  - Flytter elementet og alle de resterende ett hakk lenger ut.
  - Setter inn det nye elementet på rett plass.

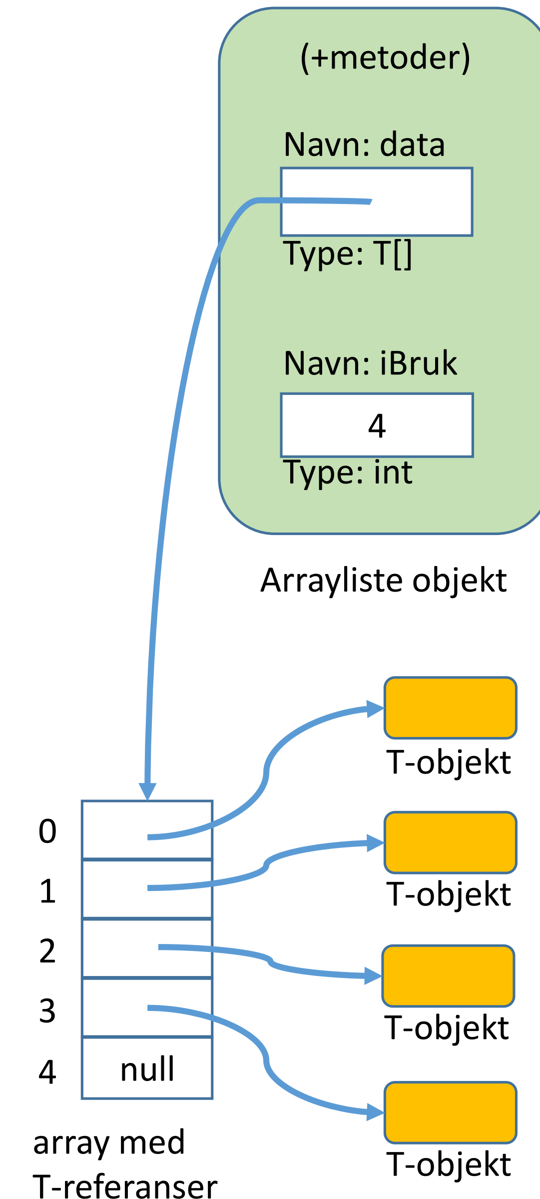
- Hvis alle elementene i listen er mindre eller lik det nye elementet, må det nye elementet settes bakerst.

# add-metode for prioritetskø (ved tom kø)

```
public class ArrayPrioKoe<T extends Comparable<T>> extends Arrayliste<T> {
    @Override
    public void add(T x) {
        if (size() == 0) {
            // Listen er tom, så sett inn nytt element:
            super.add(x);
            return;
        }
```

# add forts. (ikke-tom prioritetskø )

```
    for (int i = 0; i < size(); i++) {
        if (get(i).compareTo(x) > 0) {
            // Vi har funnet et element som er større enn det nye.
            // Flytt det og etterfølgende elementer ett hakk bak.
            super.add(null); // Utvid arrayen (overskrives straks)
            for (int ix = size()-2; ix >= i; ix--)
                set(ix+1, get(ix));
            // Sett inn det nye elementet:
            set(i, x);
            return;
        }
    }
    // Det nye elementet er størst (ellers ville vi returnert)
    // og skal inn bakerst:
    super.add(x);
} // end of add
```



# Tester add for prioritetskø

```
public class TestArrayPrioKoe {
    public static void main(String[] args) {
        Liste<Integer> ap = new ArrayPrioKoe<>();
        ap.add(4);
        ap.add(0);
        ap.add(16);
        ap.add(5);
        ap.add(2);

        ap.remove(0);

        for (int i=0; i<ap.size(); i++) {
            System.out.println(ap.get(i));
        }
    }
}
```

**compareTo** i **Integer**-klassen kalles i **add**

```
2
4
5
16
```

# Prioritetskø: Forutsetninger/ valg

- Bruker metoden **compareTo** i interface **Comparable**
  - Hver klasse kan bare ha én **compareTo** metode
  - Hvis denne ikke passer med prioritet må vi finne andre løsninger (ikke IN1010-pensum)

- For vår klasse ArrayPrioKo er det en invariant at køen alltid er sortert (som her) – da slipper vi å lete gjennom listen etter høyeste prioritet hver gang

# Å gå gjennom en Liste

- Vi kan gå gjennom en liste ved å hente elementene ett for ett med en indeks-basert for-løkke

```
for (int i = 0; i < lx.size(); i++) {
    System.out.println(lx.get(i));
}
```

- Men Java har også en for-løkke som ligner den vi har brukt i Python

```
for (String elem: lx) {
    System.out.println(elem);
}
```

- Kalles ofte *enhanced*, i IN1010 *spesialisert* for-løkke. Eller bare *for-each*

=> Hva skal til for å kunne bruke denne?

# Java interface **Iterable**

- En spesialisert for-løkke itererer gjennom elementene i en beholder som implementerer interface **Iterable**

- interface **Iterable** krever bare at beholderen tilbyr én ekstra metode

| | |
|---|---|
| `Iterator<T>` | `iterator()`<br>Returns an iterator over elements of type T. |

(klippet fra Java API, java.lang)

- Men hva er en **Iterator<T>** som denne metoden skal returnere?

# Javas interface **Iterator**

- En iterator kan gi oss elementene i en beholder ett for ett, holde orden på hvor langt vi er kommet i gjennomgangen og sjekke om det er flere elementer igjen

```
public interface Iterator<T> {
    boolean hasNext();
    T next();
}
```

interface **Iterator** ligger i Java API (java.util)

- Vi må med andre ord lage en egen Iterator-klasse for Arrayliste som tilbyr disse metodene for Arrayliste beholdere

- **iterator**-metoden i **Arrayliste** kan så returnere et objekt av denne **Iterator**-klasen

# Utvider Liste-interfacet med **Iterable**

```
interface Liste<T> extends Iterable<T>{
    int size();
    void add(T x);
    void set(int pos, T x);
    T get(int pos);
    T remove(int pos);
}
```

- Alle Liste-implementasjoner må nå være itererbare – dvs tilby metoden **iterator** som returnerer et **Iterator**-objekt for beholderen.
- **Iterator-objektet** må tilby **next** og **hasNext** for beholder-objektet sitt

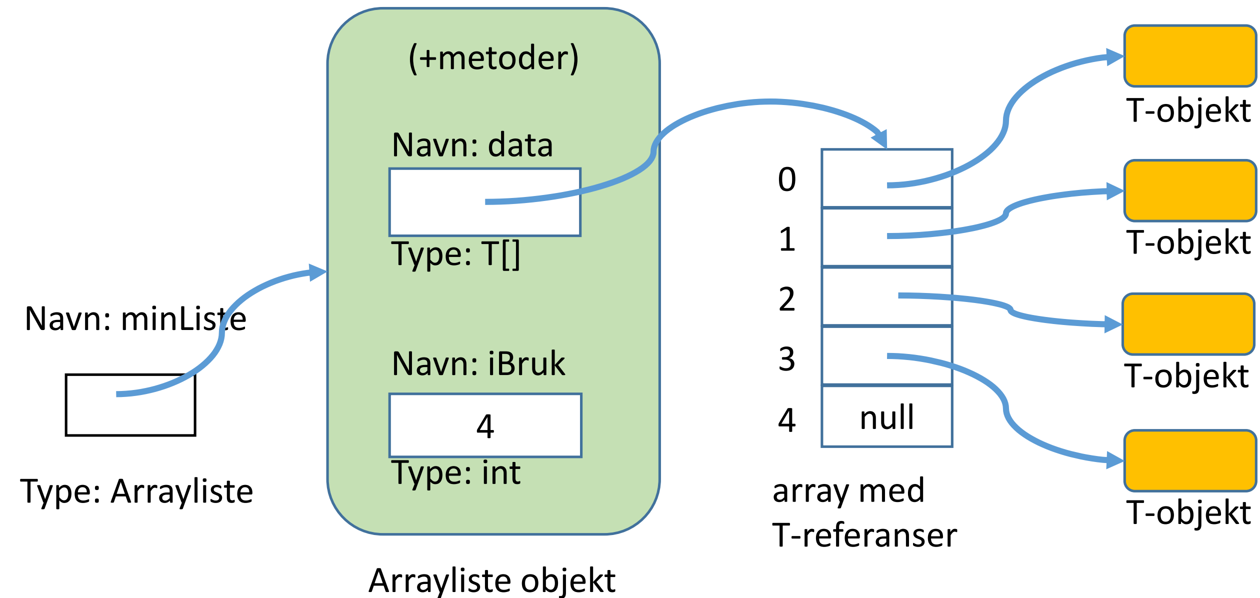
# Implementerer dette i Arrayliste-klassen

- Lager en klasse ListeIterator som en indre klasse i ArrayListe
- Da har den tilgang på alle egenskaper i ArrayListe (metoder og instansvariabler)
- Trenger ikke en egen klassevariabel for type – den skal alltid brukes for en bestemt beholder, der elementene er av type T

```
public class Arrayliste<T> implements Liste<T>{
    // instansvariabler

    class ListeIterator implements Iterator<T> {

        @Override
        public T next() { //innhold
        }

        @Override
        public boolean hasNext() { // innhold
        }
    }

    // Andre metoder
```

# Implementerer dette i Arrayliste-klassen

- Superklassen Arrayliste inneholder datastrukturen for alle subklassene
- Invarianter for alle Arrayliste beholderne:
  - iBruk angir antall elementer
    - iBruk > 0 betyr at beholderen inneholder minst 1 element
  - Elementer i ikke-tom beholder er alltid i riktig rekkefølge
    - data[0] er neste som tas ut
    - data[iBruk-1] er siste som skal tas ut
- Vi kan derfor lage en Iterator for Arrayliste som virker for alle subklassene



# Klassen ListeIterator i ArrayListe

```
public class Arrayliste<T> implements Liste<T>{
    // instansvariabler
     class ListeIterator implements Iterator<T> {
         private int pos = 0;
         @Override
         public T next() {
             pos++;
             return get(pos-1);
         }
         @Override
         public boolean hasNext() {
             return pos < size();
         }
     }
    // Andre metoder
```

- Må lage et nytt Iterator-objekt før hver gjennomgang av elementene i en beholder
- Trenger en posisjonspeker som holder rede på hvor langt vi er kommet (instansvariabel **pos**)
- Bruker metodene get og size i ArrayListe (kunne også aksessert instansvariablene **iBruk** og **data** direkte)

# Metoden **iterator()** i **Arrayliste**

```
@Override
public Iterator<T> iterator() {
    return new ListeIterator();
}
```

- ListeIterator-objektet som returneres har tilgang til egenskapene i det Arrayliste-objektet metoden ble kalt på

- Java kan nå tilby **for-each** løkker på Arrayliste-beholdere, fordi vi tilbyr et Iterator-objekt med **next-** og **hasNext-** metoder.

- (Programmerere som bruker Arrayliste-beholderen vår kan også be om et iterator-objekt for å kalle **next** og **hasNext** på sin beholder - må importere java.util.Iterator)

# Itererbar Arrayliste – endret:

- Liste<T> extends Iterable<T>
  - Utvidet med metoden **iterator**
- ArrayListe
  - implementerer metoden **iterator**
  - .. som oppretter Iterator-objekt for Arrayliste
  - .. fra den indre klassen ListeIterator som tilbyr **hasNext-** og **next**-metoder
- TestArrayliste kan nå
  - bruke for-each (Java oppretter og bruker Iterator-objekt bak kulissene) - eller
  - lage et eget Iterator-objekt for å hente ut elementer i rekkefølge (må da importere java.util.Iterator)

```java
import java.util.Iterator;
public class Arrayliste<T> implements Liste<T>{
    @SuppressWarnings("unchecked")
    private T[] data = (T[]) new Object[10];
    private int iBruk = 0;

    class ListeIterator implements Iterator<T> {
        private int pos = 0;
        @Override
        public T next() {
            pos++;
            return data[pos-1];
        }
        @Override
        public boolean hasNext() {
            return pos < iBruk;
        }
    }

    @Override
    public Iterator<T> iterator() {
        return new ListeIterator();
    }

    // Andre metoder
```

# Tester Arrayliste med iterator

```
import java.util.Iterator;

public class TestArrayliste<T> {
    public static void main(String[] args) {
        Liste<String> lx = new Arrayliste<>();
        // Sett inn 13 elementer:
        for (int i = 0; i < 13; i++) {
            lx.add("A" + i);
        }
        // test av metodene fra forrige uke (kode på uke7-siden)

        for (String elem: lx) {
            System.out.println(elem);
        }

        System.out.println("Tester eksplisitte kall på iterator:");
        Iterator<String> iter = lx.iterator();
        while (iter.hasNext()) {
            System.out.println(iter.next());
        }
    }
}
```

# Hva med beholderne som arver Arrayliste?

```
public class TestArrayPrioKoe {
    public static void main(String[] args) {
        Liste<Integer> ap = new ArrayPrioKoe<>();
        ap.add(4);
        ap.add(0);
        ap.add(16);
        ap.add(5);
        ap.add(2);

        ap.remove(0);

        System.out.println("Tester iterable");
        for (Integer x: ap) {
            System.out.println(x);
        }
    }
}
```

```
>java TestArrayPrioKoe
Tester iterable
2
4
5
16
```

# Oppsummering

- Flee mulige datastrukturer for lenkelister

- Har beskrevet og delvis implementert lister med ulik semantikk
  - Kø (setter inn bakerst)
  - Stabel (setter inn foran)
  - Prioritetsliste (setter inn etter prioritet)

- Mer Java
  - Innpakking ("boxing")
  - Å sammenligne objekter (Comparable )
  - Å gå gjennom alle elementer i en beholder (vha Iterator)

- Neste uke repetisjon: Gjennomgang av tidligere eksamensoppgave v/ Dag Langmyhr