Föreläsning 4

Innehåll

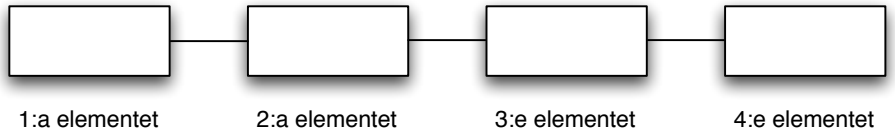
- Abstrakta datatypen **lista**
  - listklasser i Java,
  - egen implementering
- Datastrukturen **enkellänkad lista**
- Nästlade klasser
  - statiskt nästlade klasser
  - inre klasser

Abstrakta datatypen lista

Definition

En lista är en följd av element.

- Det finns en före-efter-relation mellan elementen.
- Begrepp som "första elementet i listan", "efterföljaren till visst element i listan" är meningsfulla. Det finns alltså ett positionsbegrepp.
- Definitionen innebär *inte* att elementen är sorterade på något visst sätt t.ex. i storleksordning.



Abstrakt datatypen lista

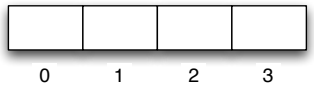
Abstrakt datatyp

En abstrakt modell tillsammans med de operationer man kan utföra på den.

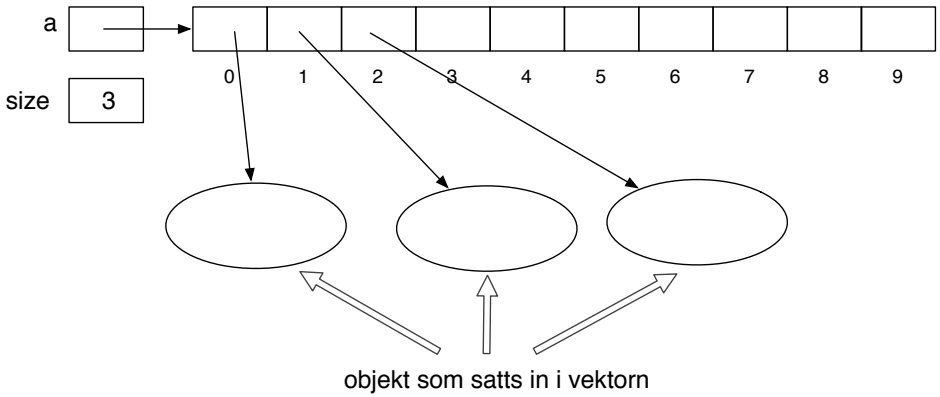
- Abstrakt modell: lista
- Operationer på modellen:
  - Lägga in element i listan (först, sist ...)
  - Ta bort ett element ur listan
  - Undersöka om ett visst element finns i listan
  - Ta reda på ett elementet i listan (första, sista ...)
  - Undersöka om listan tom
  - ...

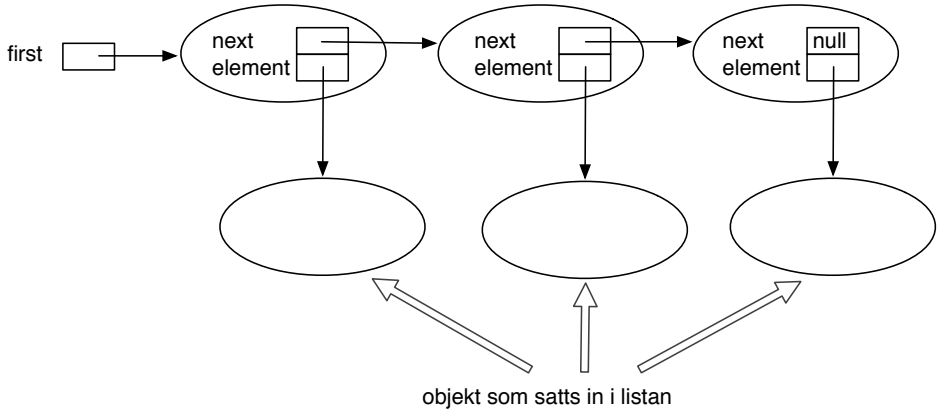
Implementering av listor

- En vektor kan användas för att hålla reda på listans element.



- Ett annat sätt är att utnyttja länkad datastruktur.
  - I en länkad struktur består listan av noder som har en referens till efterföljaren (och ev. till föregångaren).






```
public class SingleLinkedList<E> {
   private ListNode<E> first; // referens till första noden
                              // null om listan är tom

   ..metoder..

   /* Statisk nästlad klass. Representerar en nod som
      innehåller ett element av typ E. */
   private static class ListNode<E> {
      private E element;        // data som lagras
      private ListNode<E> next; // refererar till nästa nod

      private ListNode(E e) {
         element = e;
         next = null;
      }
   }
}
```


Klasser kan deklareras *inuti* andra klasser (**nästlade klasser**).

- Används oftast när den nästlade klassen bara är meningsfull för den omgivande klassen.
- Användare behöver oftast inte känna till existensen av den nästlade klassen.
- En nästlad klass kan deklareras `private` om den bara ska användas i den omgivande klassen. Även konstruktorn kan då vara `private`.
- I den omgivande klassen har man tillgång till allt i den nästlade klassen (även det som är `private`).

- Det finns två typer av nästlade klasser:
  - **statiska nästlade klasser**
  - **inre klasser** (eng: inner classes).

Statiska nästlade klasser

```
public class OuterClass {
   ...

   public void p() {
      NestedClass x = new NestedClass();
      ...
   }

   private static class NestedClass {
      private NestedClass() {...}
      ...
   }
}
```

En statisk nästlad klass kan bara komma åt statiska attribut och statiska metoder i den omgivande klassen.

Inre klasser

```
public class OuterClass {
   private int i;

   public void p() {
      InnerClass x = new InnerClass();
      ...
   }

   private class InnerClass {
      private InnerClass() {...}

      private void q() {
         int b = i; ...;   // Här används i från OuterClass!
      }
   }
}
```

Ett objekt av en inre klass kan komma åt allt i det objekt av den omgivande klassen som skapade objektet av den inre klassen.

Att skapa objekt av nästlade klasser

- Görs oftast bara i den omgivande klassen.
  - Då blir det samma syntax som vanligt.
  - Exempel finns på föregående bilder.
- Man kan skapa objekt av nästlade klasser även utanför den omgivande klassen.
  - Kräver dock att den nästlade klassen och dess konstruktor är `public`.
  - Detaljer på nästa bild.

Att skapa objekt av nästlade klasser

Statiska nästlade klasser

Om den nästlade klassen är *statisk*:

```
public class OuterClass {
   ...

   public static class NestedClass {
      public NestedClass() {...}
      ...
   }
}
```

så skapas en instans av den nästlade klassen med följande syntax:

```
OuterClass.NestedClass x = new OuterClass.NestedClass(...);
```

Att skapa objekt av nästlade klasser

Inre klasser

Om den nästlade klassen är en *inre* klass:

```
public class OuterClass {
   ...

   public class InnerClass {
      public InnerClass(...) {...}
      ...
   }
}
```

så kan instanser av den inre klassen bara skapas genom ett objekt av den yttre klassen:

```
OuterClass a = new OuterClass();
OuterClass.InnerClass b = a.new InnerClass();
```


Inre klass eller statisk nästlad klass i vår listklass?

- Det fungerar alltid med en inre klass.

```
public class SingleLinkedList<E> {
   ...
   private class ListNode { ...
```

  - Men varje objekt av den inre klassen har en referens till ett objekt av den omgivande klassen.
  - Dessa referenser tar upp minne.

- Om man i den nästlade klassen bara behöver använda sådant som är deklarerat static i den omgivande klassen kan man istället ha en statisk nästlad klass.

```
public class SingleLinkedList<E> {
   ...
   private static class ListNode<E> { ...
```

  - I den statisk nästlade klassen `ListNode` når vi inte typparametern i den omgivande klassen. `ListNode` måste därför också ha en typparameter.

Exempel på metoder i en enkellänkad lista

Insättning och borttagning först i listan

Länka in en ny nod innehållande elementet x *först* i listan:

```
public void addFirst(E x) {
   ListNode<E> n = new ListNode<E>(x);
   n.next = first;
   first = n;
}
```

Tag bort första noden i listan, returnera dess innehåll:

```
public E removeFirst() {
   if (first == null) {
      throw new NoSuchElementException();
   }
   ListNode<E> temp = first;
   first = first.next;
   return temp.element;
}
```


Traversering av elementen i listan

Exempel: metoden `toString`

Returnera en sträng som representerar listan:

```
public String toString() {
    StringBuilder sb = new StringBuilder();
    sb.append('[');
    ListNode<E> p = first;
    while (p != null) {
        sb.append(p.element.toString());
        if (p.next != null) {
            sb.append(", ");
        }
        p = p.next;
    }
    sb.append(']');
    return sb.toString();
}
```

Traversering av elementen i listan

Mönster

```
ListNode<E> p = first;
while (p != null) {
    ...
    p = p.next;
}
```


Diskutera

Antag att vi ska skriva metoder `addLast` och `removeLast` för att sätta in och ta bort sist listan.

Hur ska vi lösa de problemen?
Vilka specialfall finns?

Exempel på metoder i en enkellänkad lista

Insättning sist i listan

Länka in en ny nod innehållande elementet `x` `sist` i listan:

```
public void addLast(E x) {
   ListNode<E> n = new ListNode<E>(x);
   if (first == null) {
      first = n;
   } else {
      ListNode<E> p = first;
      while(p.next != null) {
         p = p.next;
      }
      p.next = n;
   }
}
```


Söka upp sista noden i listan

Mönster

```
if (first == null) {
    ...
} else {
    ListNode<E> p = first;
    while (p.next != null) {
        p = p.next;
    }
    // Här refererar p till sista noden
}
```

Exempel på metoder i en enkellänkad lista

Borttagning sist i listan

```
public E removeLast() {
    if (first == null) {        // tom lista
        throw new NoSuchElementException();
    }
    if (first.next == null) {   // ett element
        ListNode<E> temp = first;
        first = null;
        return temp.element;
    }
    ListNode<E> p = first;      // minst två element
    ListNode<E> pre = null;
    while (p.next != null) {
        pre = p;
        p = p.next;
    }
    pre.next = null;
    return p.element;
}
```

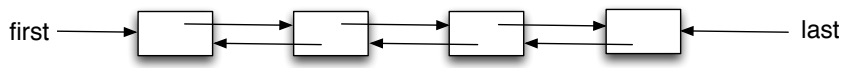

Diskutera

- Två av metoderna vi implementerat, `addlast` och `removeLast`, är långsammare än motsvarande metoder för att sätta in och ta bort i början av listan. Både `addlast` och `removeLast` innehåller en loop.
- Ge förslag på hur man kan implementera listklassen så att dessa loopar kan tas bort.

Implementering med länkad struktur

Kommentarer

- Exemplen visar att det är viktigt att tänka på specialfall.
- Vissa operationer blir krångliga i den enkellänkade implementeringen.
  - Dessa kan förenklas om man i varje nod också har en referens till föregångaren. Detta kallas dubbellänkade listor.



  - Man kan förenkla implementeringar av vissa operationer ytterligare genom att ha ett speciellt element ("huvud") i början av listan.

Traversering av listor – iteratorer

Användare av en listklass behöver möjlighet att gå igenom elementen i listan. Låt därför `SingleLinkedList` implementera interfacet `Iterable`:

```
public class SingleLinkedList<E> implements Iterable<E>  {
   ...
```

- Lägg till metoden

  `Iterator<E> iterator()`

  i klassen `SingleLinkedList`. Metoden `iterator` ska skapa och returnera ett iterator-objekt.

- Skriv en (inre) klass som implementerar interfacet `Iterator` enligt mönstret:

  `private class MyListIterator implements Iterator<E> {...}`

Interfacet Iterator

metoder

```
/** Returns true if the iteration has more elements. */
boolean hasNext();

/** Returns the next element in the iteration. */
E next();

/** Removes from the underlying collection the last
    element returned by the iterator (optional). */
default void remove();
```

Metoder deklarerade `default` är redan implementerade. Default-metoden `remove` genererar `UnsupportedOperationException`.

Iteratorklass

Inre klassen `MyListIterator`

```
private class MyListIterator implements Iterator<E> {
   private ListNode<E> pos;

   private MyListIterator() {pos = first;}

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

   public E next() {
      if (hasNext()) {
         ListNode<E> temp = pos;
         pos = pos.next;
         return temp.element;
      } else {
         throw new NoSuchElementException();
      }
   }
   ...
```


Metoden `iterator()`

Klassen `SingleListIterator`

```
public class SingleLinkedList<E> implements Iterable<E>  {
   private ListNode<E> first;
   ...

   public Iterator<E> iterator() {
      return new MyListIterator();
   }

   private class MyListIterator implements Iterator<E> {...}
   ...
}
```


Användning av iterator

Exempel

- Nu kan vi iterera genom vår lista:

```
SingleLinkedList<String> list = new SingleLinkedList<String>(
// sätt in några String-objekt i listan

...
Iterator<String> itr = list.iterator();
while (itr.hasNext()) {
   String s = itr.next();
   ...
}
```

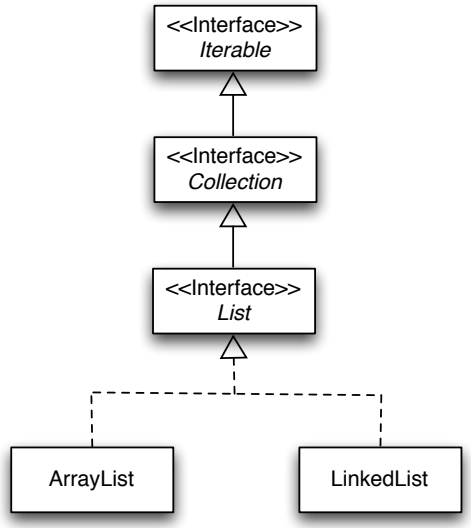
- Eftersom vår klass `SingleLinkedList` implementerar interfacet `Iterable` kan vi också använda "foreach"-satsen:

```
for (String s : list) {
   ...
}
```

Listklasser i java.util

Det finns två konkreta generiska klasser i Javas API för listhantering. Båda implementerar interfacet `List`.

- `ArrayList<E>`, som implementerats med vektor
- `LinkedList<E>`, som implementerats med en dubbel-länkad cirkulär struktur

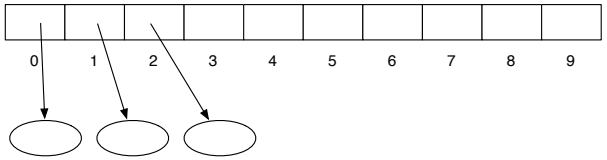


Listklasser i java.util

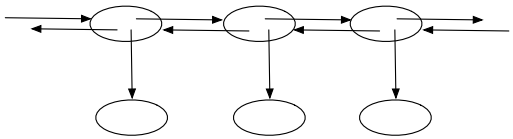
Implementeringar

Nu när vi implementerat en egen listklass kan vi jämföra olika listimplementeringar:

- `ArrayList` implementeras med hjälp av en vektor:



- Inuti `LinkedList` används en cirkulär, dubbellänkad struktur:



Fördelar, nackdelar?

Diskutera

Kryssa i tabellen vilka metoder som är snabba (konstant tid) respektive långsammare (innehåller loop, beror på antal element i listan).

		snabb	långsammare
ArrayList:	get(i)		
	addFirst(obj)		
	addLast(obj)		
	remove(i)		
LinkedList:	get(i)		
	addFirst(obj)		
	addLast(obj)		
	remove(i)		

ArrayList vs LinkedList

- `ArrayList`
  - De indexerade metoderna `get(int idx)` och `set(int idx, E element)` är effektiva i `ArrayList`.
  - Däremot är insättningar och borttagningar (utom sist i listan) långsamma eftersom element måste flyttas.
- `LinkedList`
  - De indexerade metoderna är långsamma eftersom listan måste stegas igenom tills önskat element nås.
  - När man väl hittat rätt plats i listan är insättningar och borttagningar snabba.
  - Nod-objekten kräver extra minne och hantering.

- Implementera en lista effektivt med hjälp av vektor respektive länkade struktur.
- Förklara vad nästlade och inre klasser är för något samt kunna implementera sådana.
- Med hjälp av dokumentation använda klasser och interface från Java Collections Framework: `List`, `Queue`, `Deque`, `ArrayList`, `LinkedList`, `ArrayDeque`, `Iterator`, `ListIterator` och `Iterable`

- Räkna hur många gånger olika ord förekommer i en text. – Använd en map för att hålla reda på ord och deras frekvens.
- I programmet ska du använda interface och klasser från Java Collection Framework.
  - Tips! Se Javas dokumentation för nätet för de klasser du använder.
- Innehåll: abstrakta datatyperna lista, mängd, map. Använda generiska klasser.