Clase 25

Listas enlazadas

Colección de clases de Java

- El paquete `java.util` contiene implementaciones de muchas de las estructuras de datos que vamos a tratar y que implementaremos de forma más sencilla en clase.
- Puede utilizar las implementaciones estándar de Java en los boletines de problemas. Están más elaboradas y son más abstractas que las implementaciones de clase.
- Al aprender a implementar varias estructuras de datos, profundizará en el modo en que utiliza todas las estructuras.

Listas como un tipo de datos abstracto

Una *lista* es un conjunto de elementos con un orden concreto:

- **Puede tener una longitud arbitraria.**
- **Ofrece la posibilidad de insertar o eliminar un elemento en cualquier ubicación.**
- **Ofrece la posibilidad de recorrer la lista de forma ordenada, de elemento en elemento.**

3

Una interfaz `List`

```
public interface List {
  public boolean isEmpty();
  public void addFirst( Object o );
  public void addLast( Object o );
  public boolean contains(Object o);
  public boolean remove(Object o);
  public Object removeFirst()
    throws NoSuchElementException;
  public void clear();
  public int size();
}
```

4

Estrategias de implementación de listas

- Existen muchas formas de implementar una lista.
- En implementaciones basadas en *arrays* como el `Vector` de Java, insertar un elemento en cualquier lugar que no sea el final de la lista puede ser complejo, ya que todos los elementos que se encuentran entre el punto de inserción y el final de la lista deberán desplazarse una posición para dejar hueco para la nueva entrada. Ocurre algo similar con la eliminación.
- Por ello, las listas suelen utilizar una implementación *enlazada*.

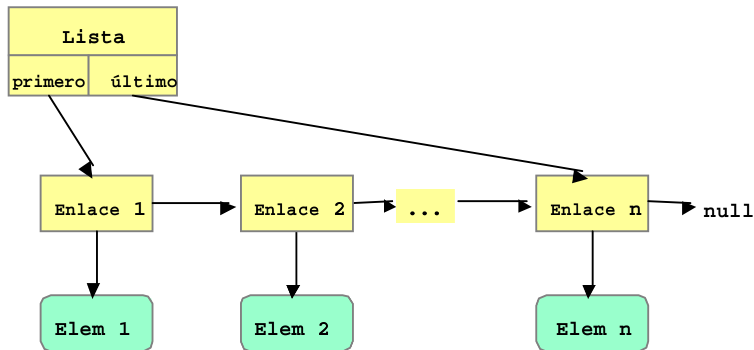
5

Listas enlazadas sencillas

- Las listas enlazadas son como trenes de mercancías.
- Cada elemento que que se va a poner en la lista está contenido en una instancia de un nuevo tipo de objeto, llamado *enlace*, que equivale a un vagón del tren.
- En una lista enlazada sencilla, el enlace no sólo contiene el elemento de la lista, sino que también apunta al siguiente elemento de la lista, al igual que un vagón de mercancías está acoplado al siguiente.
- El último enlace de la lista no apunta a nada.

6

Diagrama de lista enlazada sencilla



7

Listas enlazadas sencillas, 2

- El objeto de la lista, en sí mismo, apunta al enlace que contiene el primer elemento mostrado y suele incluir una referencia adicional al último enlace de la lista para facilitar la incorporación de elementos.
- Se dice que un enlace "*contiene*" o "*apunta a*", y que la instancia de la lista "*apunta*" o "*contiene un puntero*". En Java, todos ellos son sinónimos de contener una referencia.
- Así, el enlace realmente no contiene el elemento, sino una referencia que señala al elemento de la lista.
- El último enlace contiene una referencia *null* en el campo que apunta al siguiente elemento.

8

Demostración de una lista enlazada sencilla

`List:` *Interfaz de lista*

`SLinkedList:` *Implementación de lista*

`SLinkedListApp:` *Aplicación main()*

`SLinkedListView:` *GUI de lista*

9

La clase interna de enlace

```
public class SLinkedList implements List {
  private static class SLink
  {
    Object item;
    SLink next;

    SLink( Object o, SLink n )
    { item = o; next = n; }

    SLink( Object o )
    { this( o, null ); }
  }
  . . .
```

10

Listas genéricas y tipadas

- **La interfaz `List` que hemos especificado es general, como la clase `Vector` de Java: almacena y recuperar objetos.**
- **Si crea su propia clase de tipo lista y sabe, por ejemplo, que sólo trabajar á concadenas, puede sustituir los campos `Object` por campos `String`. Por ejemplo:**

```
private static class SLink {
  String item; SLink next;
  SLink( String o, SLink n ) { item = o; next = n; }
  SLink( Object o )          { this( o, null ); }
}
public void addFirst( String o );
```

11

Miembros de datos de `SLinkedList`

- **Sólo es necesario `first` (primero).**
- **`last` (último) y `length` (longitud) se pueden encontrar al recorrer la lista, pero si se conservan y se actualizan estos miembros, la llamada a `size()` y a `append()` es mucho más rápida.**

```
private int length = 0;
private SLink first = null;
private SLink last = null;
```

12

== y el método `Object equals`

- `contains( Object o )` **y** `remove( Object o )` **deben buscar** `Object o` **en la lista. Pero, ¿qué implica *encontrarlo*?**
- **¿Debe contener la lista una referencia al objeto idéntico (==)? ¿O basta con que contenga una referencia a un objeto *equivalente pero posiblemente* distinto?**

```
static private boolean
   objectEquals( Object a, Object b )
{
       if ( a == null )
        return ( b == null );
       else
        return a.equals( b );
}
```

13

Atención a los casos especiales

- **Lo difícil al implementar una lista enlazada no es implementar el caso habitual para cada método, por ejemplo, eliminar un objeto de la mitad de la lista.**
- **Lo complicado es comprobar que los métodos también funcionen en casos excepcionales y ambiguos.**
- **Para cada método, debe pensar en si la implementación funcionará en los siguientes casos**
  - **en una lista vacía,**
  - **en una lista con uno o dos elementos,**
  - **en el primer elemento de una lista,**
  - **en el último elemento de una lista.**

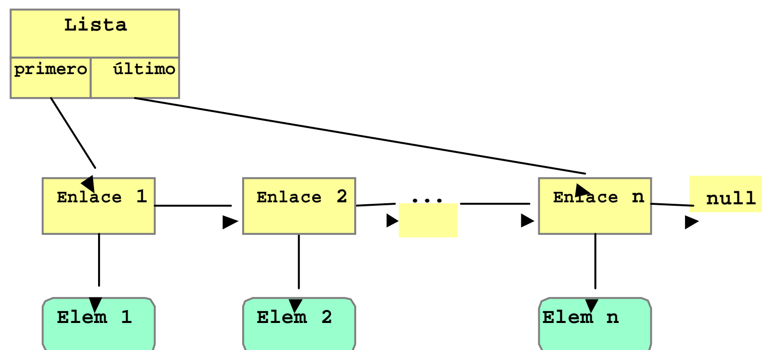
14

removeFirst()

```
public Object removeFirst()
   throws NoSuchElementException
{
    if ( first == null )    // si la lista está vacía
        throw new NoSuchElementException();
    else {
        SLink t = first;
        first = first.next;
         // si la lista tenía 1 elemento y ahora está vacía
        if ( first == null ) last = null;
        length--;
        return t.item;
    }
}
```

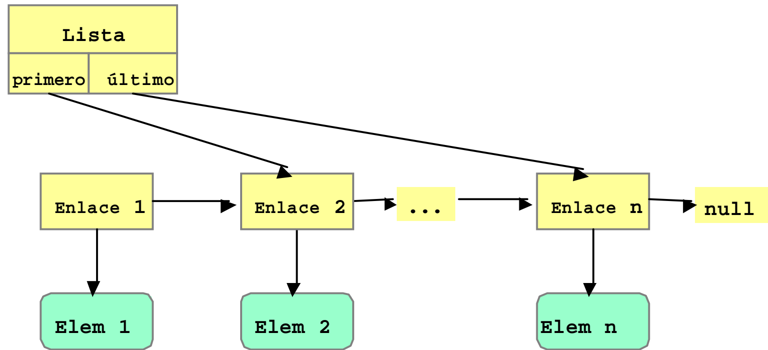
15

removeFirst(), antes



16

`removeFirst()`, después

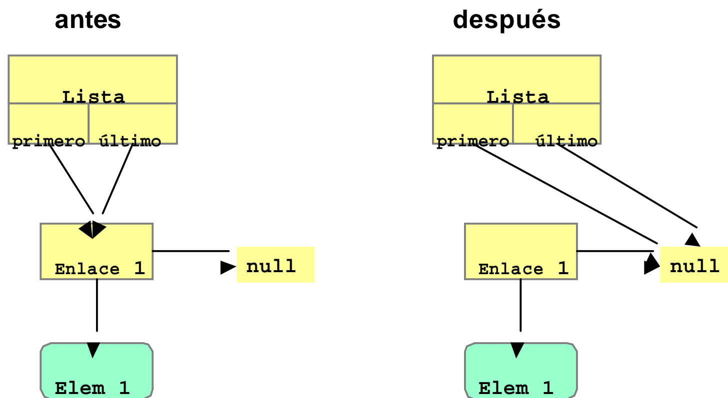


17

`removeFirst()`, caso especial

antes

después



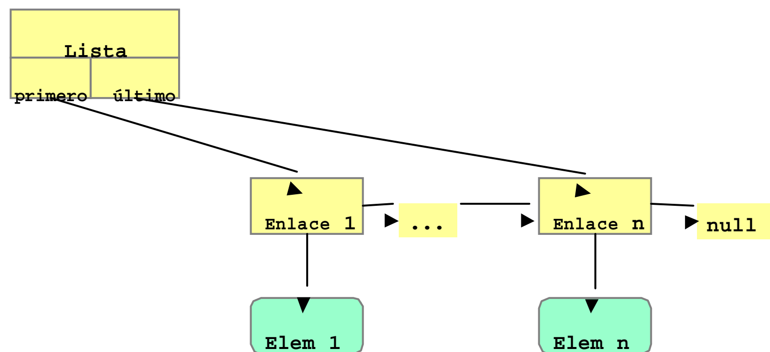
18

addFirst(Object o)

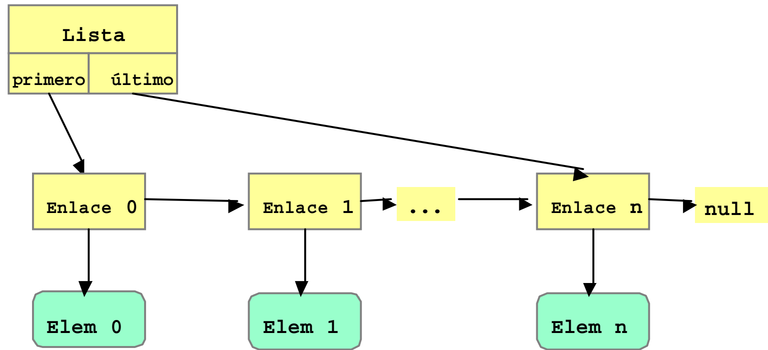
```
public void addFirst(Object o)
{
    if ( first == null )    // si la lista está vacía
    {
        first = last = new SLink( o );
    }
    else
    {
        first = new SLink( o, first );
    }

    length++;
}
```


addFirst(), después

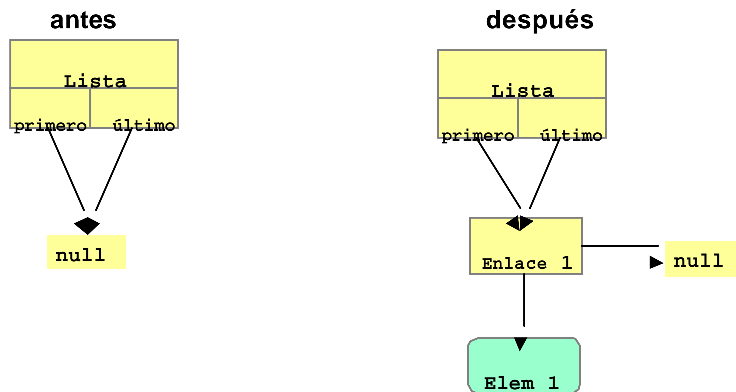


`addFirst()`, después



21

`addFirst()`, caso especial



22

Lista enlazada - Ejercicio 1.1

- Descargue el archivo `LinkedListSim.zip` del sitio web de clase. Hay un enlace a la página de material de clase.
  - En Netscape, vaya a http://web.mit.edu/1.00/www/Lectures
  - En Clase 25, pulse LinkedListSim.zip con el botón derecho y seleccione "Guardar como". Guarde el archivo en el escritorio.
- El archivo zip se descomprimirá en un directorio llamado `LinkedListSim`.
  - Haga doble clic en el archivo que ha guardado en el escritorio
  - Pulse el botón Extraer en el panel de comandos
  - Mediante el panel Carpetas/Unidades, desplácese hasta la ubicación en la que desea crear el directorio del proyecto. Pulse Extraer en el menú emergente homónimo y los archivos se descomprimirán en un subdirectorio llamado `LinkedListSim`.

Lista enlazada - Ejercicio 1.2

- En Forte, cree un nuevo proyecto llamado `LinkedListSim` y adjunte el directorio que acaba de crear.
  - En Forte, seleccione Project->Project Manager
  - Haga clic en el botón New del menú emergente Project Manager
  - Nombre el proyecto como LinkedListSim
  - En la ficha File Systems, pulse con el botón derecho para acceder a la línea Files Systems situada al principio.
  - Seleccione el directorio de montaje y desplácese hasta el directorio raíz de LinkedListDirectory que acaba de crear. Seleccione el subdirectorio LinkedListSim y pulse Mount.
- Agregue todos los archivos de Java a dicho directorio en el nuevo proyecto.
  - En la ficha Project, pulse con el botón derecho en Project LinkedListSim y seleccione Add Existing. Seleccione todos los archivos de Java del subdirectorio LinkedListSim y pulse OK.
- Compílelo y ejecútelo. `SLinkedListApp` contiene `main()`.

Lista enlazada - Ejercicio `addFirst()`

- Experimente con la simulación. Observe la implementación en `SLinkedList` y `List`. El resto de los archivos gestionan la simulación. No es necesario que los analice, salvo que quiera hacerlo por curiosidad.
- Los botones `addLast` y `remove` no funcionan, ya que se han eliminado las implementaciones del método correspondiente.
- Observe el método `addFirst()` y escriba `addLast()`. Tenga cuidado con los casos especiales (¿cuáles son?).
- Compile y pruebe el método con la simulación.

25

Lista enlazada - `remove()`

- Ahora escribiremos el método `remove()`. Es más complicado. Analice los métodos `contains()` y `removeFirst()` para hacerse una idea de cómo empezar. Para poder eliminar un elemento con `remove()`, tendrá que encontrarlo primero.
- ¿Cómo trataría el caso "normal" de eliminar un elemento de la mitad de la lista?. ¿Cómo repararía la “interrupción" de la lista?
  - Pista: tal vez quiera realizar el seguimiento de dos posiciones de la lista, `current` y `previous`.
- ¿Qué casos especiales ha encontrado?

26

Lista enlazada - `remove()`

- **Aunque tiene libertad para utilizar su propia estrategia, a continuación incluimos un esquema del método que podría utilizarse:**

```
Inicializar variables
mientras haya otro enlace
   si contiene el objeto que necesitamos eliminar
    si no estamos al principio de la lista
        eliminar elemento
       si acabamos de eliminar el último elemento
          reajustar el último
      de lo contrario estamos al principio de la lista
      eliminar elemento
       si sólo había un elemento en la lista
          reajustar el último
       ajustar tamaño
   de lo contrario ajustar referencias y avanzar al siguiente enlace
```

27

Listas y posición ordinal

- **Hay ciertas cosas evidentes que quisiéramos hacer con las listas pero que no podemos hacer utilizando sólo esta interfaz. Dos ejemplos:**
  - **¿Cómo puede observar los elementos de la lista sin eliminarlos?**
  - **¿Cómo podría insertar un elemento en la lista en cualquier posición que no sea el principio o el final?**
- **Si crea su propia clase de lista, podrá realizar estas operaciones dentro de ella, pero el enfoque no sería general.**
- **Un segundo enfoque está basado en el número o el índice de las posiciones de la lista.**

28

Listas indexadas

- **Entonces podríamos agregar dos métodos**
  - `public Object get( int n )`
    `throws NoSuchElementException;`
  - `public void insert( Object o, int n )`
    `throws NoSuchElementException;`
- **El siguiente bloque de código recorrerá una lista indexada, por ejemplo, myList:**

```
for ( int i = 0; i < myList.size(); i++ )
{
    Object o = myList.get( i );
    . . .
}
```

29

Listas indexadas, 2

- **Las listas implementadas con arrays (como Java `Vector`) suelen incluir estos métodos, ya que son fáciles de implementar.**
- **Ahora bien, la idea de utilizar un índice para acceder a los miembros de la lista puede ocasionar problemas.**
  - **Como el índice depende de la posición ordinal, cambiar cada vez que se agrega o elimina un elemento de la lista.**
  - **Si la lista no está implementada sobre una estructura de datos indexada como un array, el acceso al elemento indexado puede ser lento.**
- **En la vida real, cuando utilizamos listas grandes como directorios telefónicos, no tenemos en cuenta el índice de una entrada, sino su posición relativa.**

30

Iteradores

- Un *iteradore* es una clase de ayuda que se utiliza con un `List` o con otra clase de colección.
- Cuenta con métodos para devolver los miembros de la colección de uno en uno.
- Los iteradores también pueden implementar métodos que permitan modificar la colección con relación a la posición actual del iterador.

31

Interfaz `ListIterator`

```
public interface ListIterator
{
  public boolean hasNext();
  public Object next()
    throws NoSuchElementException;
  public void remove()
    throws IllegalStateException;
  public void add( Object o );
  public void set( Object o )
    throws IllegalStateException
}
```

32

Métodos de iteradores

- El tipo de iterador que presentamos aquí devuelve un nuevo elemento y avanza hasta el siguiente con la misma operación `next()`. No hay forma de volver atrás con esta interfaz. ListIterator de Java permite ir hacia delante y hacia atrás.
- El elemento más reciente devuelto por `next()` es el elemento actual.
- `remove()` eliminará el elemento actual de la colección subyacente . `set()` lo modificará.
- `add()` insertará un nuevo elemento tras el elemento actual y delante del elemento que se devolvería en la siguiente llamada a `next()`. Tras llamar a `add()`, el elemento insertado pasa a ser el nuevo elemento actual. Una llamada a `next()` devolverá el elemento ubicado después del insertado.
- La primera llamada a `next()` debería devolver el primer elemento de la lista.

33

El iterador y su lista subyacente

- Un iterador es un objeto basado en una colección subyacente, por lo que necesitamos dar con una forma de crear un iterador para una colección.
- Lo haremos agregando un método a nuestra interfaz `List`:

```
public ListIterator listIterator();
```

- ¿Se pueden tener 2 iteradores en la misma lista lista?

34

Cómo utilizar un iterador

```
List myList = new SLinkedList();
  . . .
ListIterator iter = myList.listIterator();
  . . .
while ( iter.hasNext() )
{
    Object o = iter.next();
    . . .
}
```

35

Un iterador en acción

Nuevo iterador

verde
rojo
violeta
naranja

`current` sin definir

Tras la 1ª llamada a `next()`

verde
rojo
violeta
naranja

`current` es verde

Tras agregar negro

verde
negro
rojo
violeta
naranja

`current` es negro

Tras 2ª llamada a `next()`

verde
negro
rojo
violeta
naranja

`current` es rojo

36

Un iterador en acción, 2

Tras llamar a `remove()`

verde
negro

`current` sin definir

violeta
naranja

Tras 3º llamada a `next()`

`current` es violeta

verde
negro
violeta
naranja

37

Lista enlazada - Ejercicio 2

- **Descargue el archivo `LinkedListIterSim.zip` del sitio web de clase. Hay un enlace en la página del material de clase.**
- **El archivo zip se descomprimirá en un directorio llamado `LinkedListIterSim`.**
- **Cree un nuevo proyecto en Forte llamado `LinkedListIterSim` y adjunte el directorio que acaba de crear.**
- **Agregue todos los archivos de Java al directorio en el proyecto.**
- **Compílelo y ejecútelo. `SLinkedListApp` contiene `main()`. La vista List ahora aparece con un nuevo botón `listIterator` que abrirá una nueva ventana con el iterador actual. La vista principal muestra la posición actual del iterador.**

38

Lista enlazada - Ejercicio 2, `doubleList()`

- La vista List principal también presenta un botón rojo llamado "double". Púlselo. No hace nada. Todavía.
- Al pulsar "double" se llama a un método de una nueva clase `ListUtil`:

```
public static void doubleList( SLinkedList l )
```

- `doubleList()` está actualmente vacío. Escriba una implementación para `doubleList()` que obtenga un iterador para la lista, l, y que doble cada `Integer` de la misma. Probablemente querrá utilizar el método `intValue()` de `Integer`.
- Compílelo y pruébelo.

39

Cuidado con los iteradores

Vamos a hacer un experimento preparado. Cree y rellene una lista. Cree un iterador para dicha lista. Llame a `removeFirst()` en la lista para eliminar el primer elemento. Ahora llame a `next()` en el iterador. ¿Qué ocurre? ¿Qué debería ocurrir?

Aunque nuestra implementación es razonablemente sólida, los iteradores asumen que se les llama desde una lista fija, es decir, que NO se garantizan resultados correctos si se modifica una lista tras la construcción del iterador utilizando cualquier otra lista o métodos de instancias de iteradores. ¿Cómo se "arregla" esto? ¿Qué significa arreglar? ¿Sería mejor tener un iterador que siempre diese resultados "correctos" o uno que arrojase excepciones si se ha modificado la lista subyacente?

40

Usos y variaciones de listas enlazadas

- **Como nuestra interfaz `List` procesa métodos `append()` y `removeFirst()`, es posible implementar una *cola* trivial encima del tipo de datos concreto `SLinkedList`.**
- **¿Cómo cambiaría la implementación si cada enlace tuviera una referencia anterior (previa) y una posterior (siguiente)? Estas listas reciben el nombre (sí, lo habrá adivinado) de *listas enlazadas dobles*. ¿Qué operaciones serán más fáciles?**