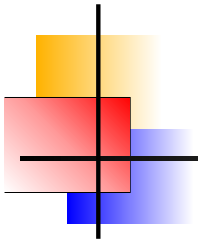


Algoritmos y Estructuras de Datos:

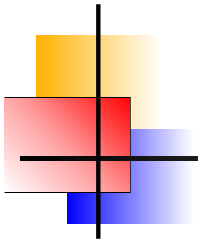
Listas de Posiciones

Guillermo Román Díez

groman@fi.upm.es

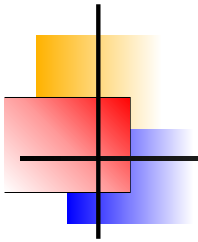
Universidad Politécnica de Madrid

Curso 2015-2016



Motivación

- Los TAD contenedores se diseñan para insertar, buscar y borrar elementos de forma eficiente.
- Una **lista** es un TAD contenedor que consiste en una **secuencia lineal** de elementos
- La *posición* de un elemento cambia al insertar y borrar
- La búsqueda de elementos *suele* ser secuencial o lineal
- No está acotada (idealmente), puede crecer de acuerdo a las necesidades del programa y de la capacidad del computador
- Un ejemplo de lista es la **lista de posiciones**
  - Implementada mediante cadenas de objetos nodo doblemente enlazadas

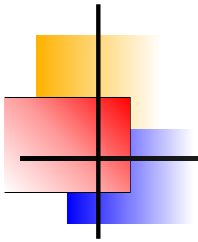


El TAD lista

- El TAD lista viene dado por un interfaz que define la funcionalidad de la lista y por una (o varias) clases que lo implementan
- Las listas se suelen implementar como cadenas enlazadas de objetos (recordad *Programación II*)

Pregunta

¿Qué operaciones tiene una lista?



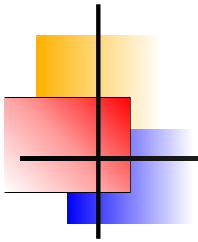
El TAD lista

- El TAD lista viene dado por un interfaz que define la funcionalidad de la lista y por una (o varias) clases que lo implementan
- Las listas se suelen implementar como cadenas enlazadas de objetos (recordad *Programación II*)

Pregunta

¿Qué operaciones tiene una lista?

- Vamos a centrarnos en el código del interfaz `PositionList<E>` para entender su funcionalidad

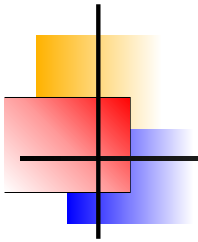


Interfaz PositionList<E> y Position<E>

- Las operaciones de `PositionList` son
  - Interrogadores: `size`, `isEmpty`
  - Acceso: `first`, `last`, `next`, `prev`
  - Inserción: `addFirst`, `addLast`, `addBefore`, `addAfter`
  - Modificación: `set`
  - Borrado: `remove`
  - Conversión: `toArray`
- `PositionList` utiliza la interfaz `Position<E>` para abstraer cada nodo de la lista
  - `Position` significa **nodo abstracto**
  - Sólo tiene la operación `element` que devuelve el elemento del nodo
- La excepción `InvalidPositionException` será lanzada cuando no se recibe una posición correcta

Ejemplo

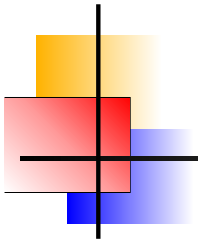
Ver los interfaces `PositionList` y `Position`



Clientes de PositionList

- No nos interesa la implementación concreta, sólo los métodos que proporciona el interfaz
- También es necesario conocer los constructores

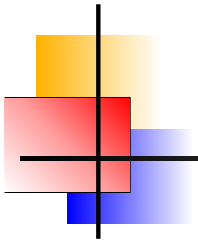
```
PositionList<String> list = new NodePositionList();
Position<String> cursor;
list.addFirst("vais");
list.addLast("a");
list.addLast("aprobar");
list.addFirst("no");
cursor = list.last();
list.set(cursor, "suspender");
cursor = list.prev(cursor);
list.remove(cursor);
cursor = list.prev(list.last());
list.set(cursor, "quiero");
```



Recorrido de las listas

- Se recorren las listas usando bucles y nodos cursor
- La inicialización consiste hacer que el cursor apunte al primer nodo de la lista
- La condición de parada depende del problema
  - Suele incluir la condición de rango (`cursor != null`)
  - Para avanzar moveremos el cursor a la siguiente posición
  - NOTA: Ojo con los posibles elementos `null`
    - Pueden saltar `NullPointerException`
    - La comparación tiene que comprobar antes que el cursor no sea null:

```
cursor != null && cursor.element().equals(x)
```



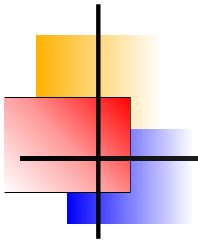
Ejemplo: Mostrar los elementos de una lista

- Con un bucle `while`

```
public static <E> void show(PositionList<E> list) {
  Position<E> cursor = list.first();
  while (cursor != null) {
    System.out.println(cursor.element());
    cursor = list.next(cursor);
  }
}
```

Ejercicio

Hacer el mismo bucle pero con `for`



Ejercicios

Ejercicio

Implementar el método `addBeforeElement`

```
public static <E> void addBeforeElement(
                                PositionList<E> list,
                                E e,
                                E a)
```

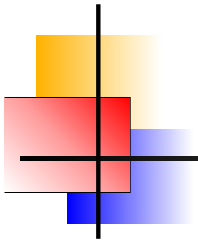
Ejercicio

Implementar el método `deleteAll`

```
public static <E> void deleteAll(
                                E elem,
                                PositionList<E> list)
```

Pregunta

¿se pueden utilizar contadores para recorrer los bucles?

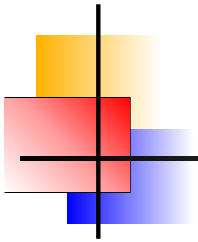


Ejercicios

Ejercicio

Implementar el método `reverseList`

```
PositionList<E> revertList (PositionList<E> list)
```



Ejercicios

Ejercicio

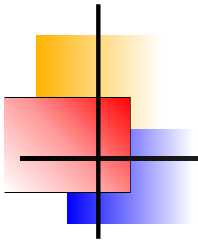
Implementar el método `reverseList`

```
PositionList<E> revertList (PositionList<E> list)
```

Ejercicio

Implementar el método `removeRange`

```
void removeRange (PositionList<E> list,
                  int posIni, int posFin)
```



Ejercicios

Ejercicio

Implementar el método `reverseList`

```
PositionList<E> revertList (PositionList<E> list)
```

Ejercicio

Implementar el método `removeRange`

```
void removeRange (PositionList<E> list,
                  int posIni, int posFin)
```

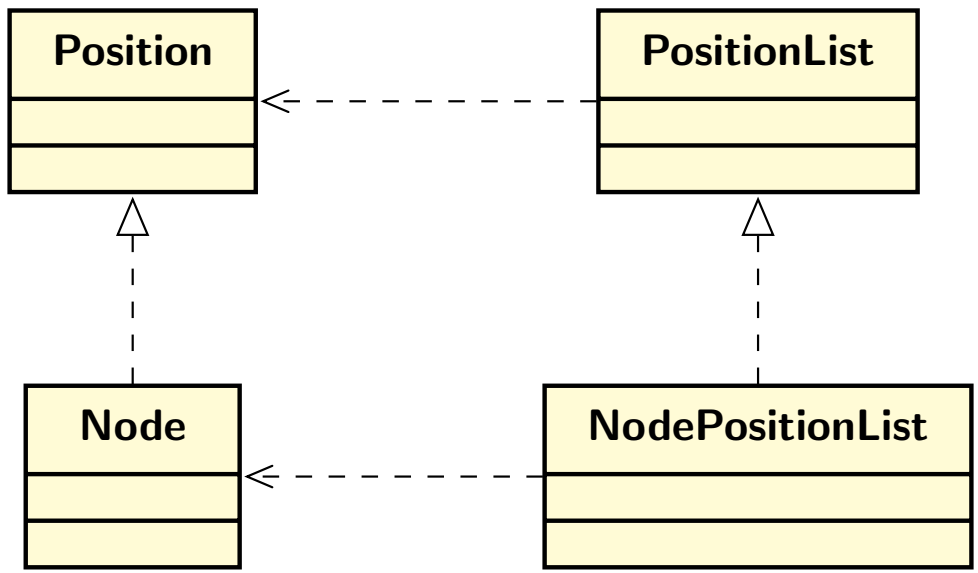
Ejercicio

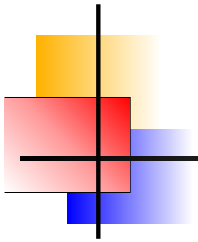
Implementar el método `trimToSize`

```
void trimToSize (PositionList<E> list, int size)
```

Implementación de Node<E> y NodePositionList<E>

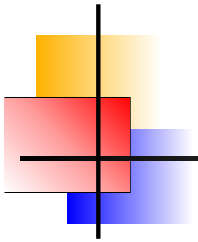
- `Node<E> implements Position<E>`
- `NodePositionList<E> implements PositionList<E>`





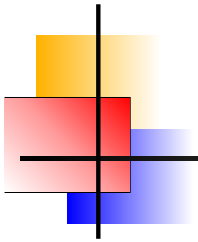
Implementación de Node<E>

- `Node<E>` implementa el interfaz `Position<E>`
- Tiene 4 atributos `owner`, `prev`, `next`, `elem`
- Tiene *getter* y *setter* para los atributos `prev`, `next`, `elem`



Implementación de Node<E>

- `Node<E>` implementa el interfaz `Position<E>`
- Tiene 4 atributos `owner`, `prev`, `next`, `elem`
- Tiene *getter* y *setter* para los atributos `prev`, `next`, `elem`
- El atributo `owner` no tiene ni getter ni setter
  - `n1.kinOf(n2)` nos indica si n1 y n2 son *parientes*
- `setPrev` y `setNext` pueden recibir `null`, ¿por qué?

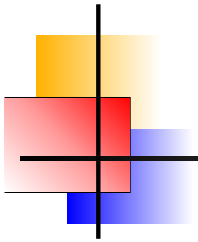


Implementación de Node<E>

- `Node<E>` implementa el interfaz `Position<E>`
- Tiene 4 atributos `owner`, `prev`, `next`, `elem`
- Tiene *getter* y *setter* para los atributos `prev`, `next`, `elem`
- El atributo `owner` no tiene ni getter ni setter
  - `n1.kinOf(n2)` nos indica si n1 y n2 son *parientes*
- `setPrev` y `setNext` pueden recibir `null`, ¿por qué?

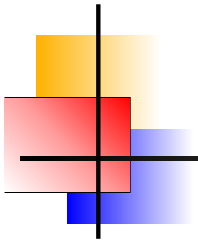
Pregunta

¿Podemos usar `Node<E>` para implementar cadenas simplemente enlazadas?



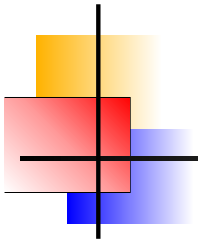
Implementación de NodePositionList<E>

- Tiene 3 atributos: el tamaño, y 2 nodos especiales: `header` y `trailer`
- Tiene 3 constructores: (1) vacio, (2) un array, (3) una lista
- Limitaciones
  - El tipo del atributo `size`
  - La memoria disponible
- El método privado `checkNode` comprueba si una posición es realmente un nodo válido
  - No es `null`
  - Es de clase `Node<E>`
  - Es un nodo de la lista (usando `kinOf` del *header* de la lista)
  - Está encadenado a un nodo previo y siguiente



Implementación de NodePositionList<E>

- El método `iayf()` devuelve un identificador único de la lista (usando `hashCode()`)
  - Se usa para identificar la lista a la que pertenece un nodo
- El método `iterator`, los métodos sobrescritos toString y `equals`, así como el método `toArray` los explicaremos cuando veamos el tema de Iteradores

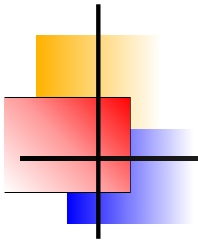


Nociones básicas de Complejidad

Complejidad

"*La complejidad de un programa es una medida de su tiempo de ejecución o del uso de memoria*"

- Hablamos de dos tipos de complejidad
  - **Complejidad temporal**: complejidad del tiempo de ejecución
  - **Complejidad espacial**: complejidad del espacio de memoria utilizado
- Las complejidades más básicas serían:
  - **O(1)**: La complejidad se mantiene constante independientemente del tamaño de la lista
  - **O(n)**: La complejidad es linealmente proporcional al tamaño de la lista



Ejercicio

Ejercicio

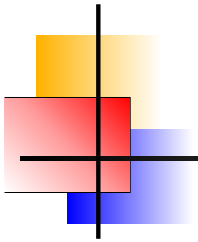
Dado un elemento, si no se tiene su posición entonces habría que buscarla. Añadir a la clase e implementar el método:

```
public Position<E> getPos(E e)
```

- que debe devolver la primera posición en la lista que contiene el elemento `e`, o `null` si el elemento no está en la lista.

Pregunta

¿Cuál sería la complejidad en el caso peor de dicho método?



Listas en Java Collections Framework

- JCF es una librería de TADs estándar de Java
- Tiene dos interfaces importantes: `Collection<E>` y `Map<K,V>`
- Muy usado en aplicaciones industriales
- Dispone de multitud de métodos útiles y con muy buena documentación
- No se puede usar en la asignatura ya que no se dispone del código fuente
- Ofrece varios TAD lista extendiendo `Collection<E>` e implementa `Iterable<E>`
  - Interfaces: `List<E>` y `ListIterator<E>`
  - Clases Abstractas Intermedias: `AbstractList<E>`, `AbstractSequentialList<E>`
  - Clases: `ArrayList<E>`, `LinkedList<E>`, `Vector<E>`, ...