Listas e Iteradores

Algorítmica y Programación II

Lic. Renato Mazzanti

Listas e implementación
Arreglos vs lista enlazada

- Dada la propia naturaleza de cada implementación de lista es difícil diseñar una abstracción única.
- Sin embargo Java define una interfaz general basada en métodos con índice.

El TAD java.util.List

- La interface incluye los siguientes métodos:

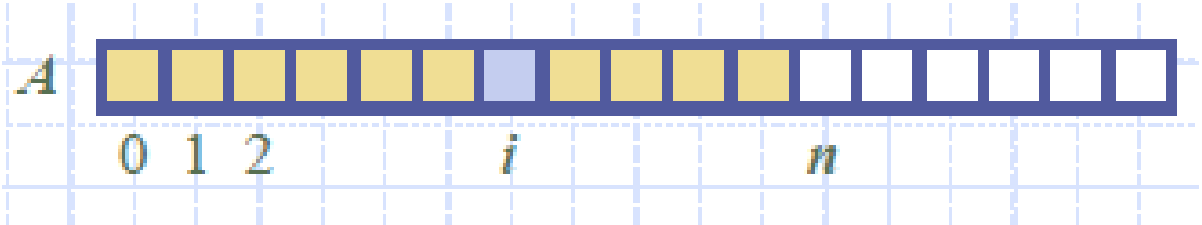
size( ):	Returns the number of elements in the list.
isEmpty( ):	Returns a boolean indicating whether the list is empty.
get(i):	Returns the element of the list having index i; an error condition occurs if i is not in range $[0, \text{size}( ) - 1]$.
set(i, e):	Replaces the element at index i with e, and returns the old element that was replaced; an error condition occurs if i is not in range $[0, \text{size}( ) - 1]$.
add(i, e):	Inserts a new element e into the list so that it has index i, moving all subsequent elements one index later in the list; an error condition occurs if i is not in range $[0, \text{size}( )]$.
remove(i):	Removes and returns the element at index i, moving all subsequent elements one index earlier in the list; an error condition occurs if i is not in range $[0, \text{size}( ) - 1]$.

Ejemplo

Method	Return Value	List Contents
add(0, A)	–	(A)
add(0, B)	–	(B, A)
get(1)	A	(B, A)
set(2, C)	"error"	(B, A)
add(2, C)	–	(B, A, C)
add(4, D)	"error"	(B, A, C)
remove(1)	A	(B, C)
add(1, D)	–	(B, D, C)
add(1, E)	–	(B, E, D, C)
get(4)	"error"	(B, E, D, C)
add(4, F)	–	(B, E, D, C, F)
set(2, G)	D	(B, E, G, C, F)
get(2)	G	(B, E, G, C, F)

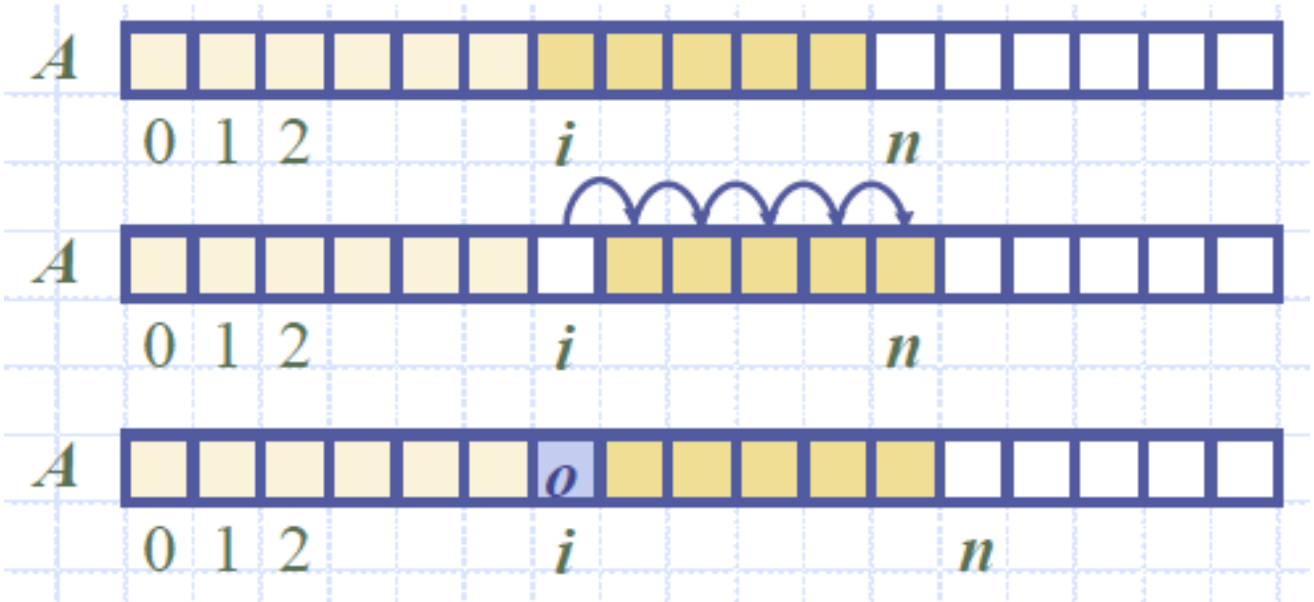
Array List

- Una opción obvia para implementar este TAD lista es usar un arreglo, A, donde A[i] almacena (una referencia a) el elemento con índice i.
- Con una representación basada en un arreglo A, los métodos get(i) y set(i, e) son fáciles de implementar accediendo A[i] (suponiendo que i es un índice legítimo).



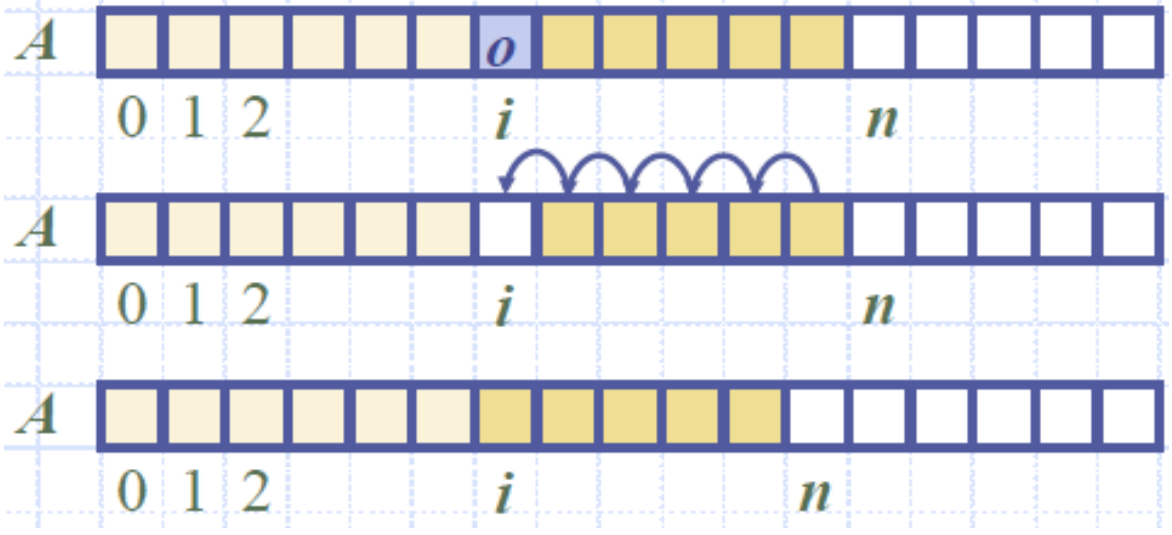
Inserción

- En la operación ***add***(***i***, ***o***), necesitamos hacer un lugar para el nuevo elemento desplazando hacia adelante los ***n*** – ***i*** elementos ***A***[***i***],…, ***A***[***n*** - 1]
- En el peor caso (***i*** = 0), toma ***O***(***n***) veces



Eliminar un elemento

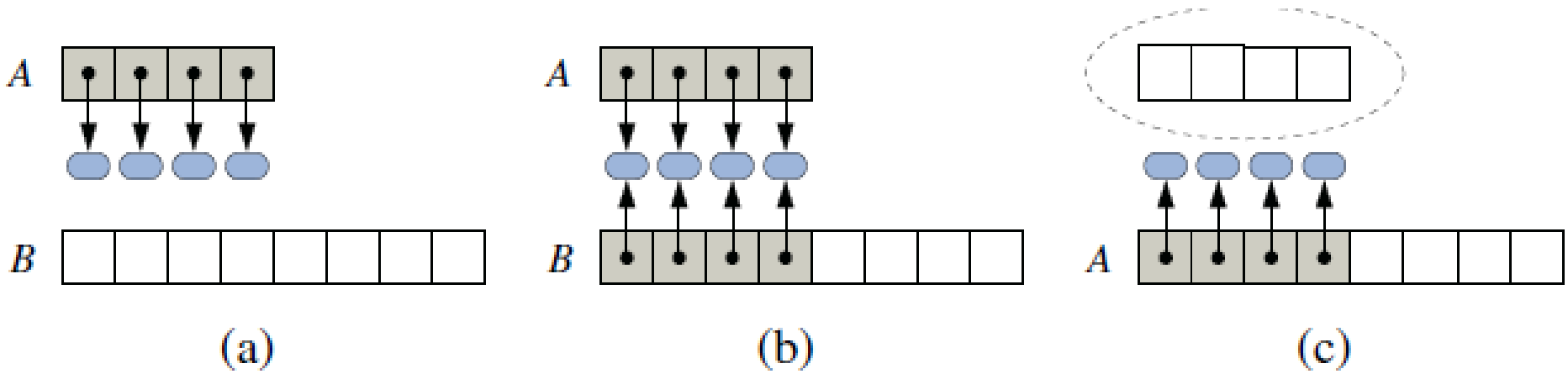
- En una operación ***remove***(i), necesitaremos llenar el hueco de la izquierda producto del elemento eliminado desplazando hacia atrás los ***n*** - ***i*** – 1 elementos ***A***[***i*** + 1], …, ***A***[***n*** - 1]
- En el peor caso (***i*** = 0), toma ***O***(***n***) veces



Performance

- Implementación de lista dinámica basada en un arreglo:
- El espacio usado por los datos es $O(n)$
- Indexar un elemento es $O(1)$
- ***add*** y ***remove*** ejecuta en $O(n)$
- En una operación ***add*** , cuando el arreglo está lleno, ***en lugar de lanzar una excepción lo podemos reemplazar por uno de mayor tamaño*** ...

Performance



Performance

Method	Running Time
size()	$O(1)$
isEmpty()	$O(1)$
get(i)	$O(1)$
set(i, e)	$O(1)$
add(i, e)	$O(n)$
remove(i)	$O(n)$

Implementación Java

```
1  public class ArrayList<E> implements List<E> {
2    // instance variables
3    public static final int CAPACITY=16;      // default array capacity
4    private E[ ] data;                        // generic array used for storage
5    private int size = 0;                     // current number of elements
6    // constructors
7    public ArrayList( ) { this(CAPACITY); }   // constructs list with default capacity
8    public ArrayList(int capacity) {          // constructs list with given capacity
9      data = (E[ ]) new Object[capacity];     // safe cast; compiler may give warning
10   }
```

Implementación Java

```
11    // public methods
12    /** Returns the number of elements in the array list. */
13    public int size( ) { return size; }
14    /** Returns whether the array list is empty. */
15    public boolean isEmpty( ) { return size == 0; }
16    /** Returns (but does not remove) the element at index i. */
17    public E get(int i) throws IndexOutOfBoundsException {
18      checkIndex(i, size);
19      return data[i];
20    }
21    /** Replaces the element at index i with e, and returns the replaced element. */
22    public E set(int i, E e) throws IndexOutOfBoundsException {
23      checkIndex(i, size);
24      E temp = data[i];
25      data[i] = e;
26      return temp;
27    }
```

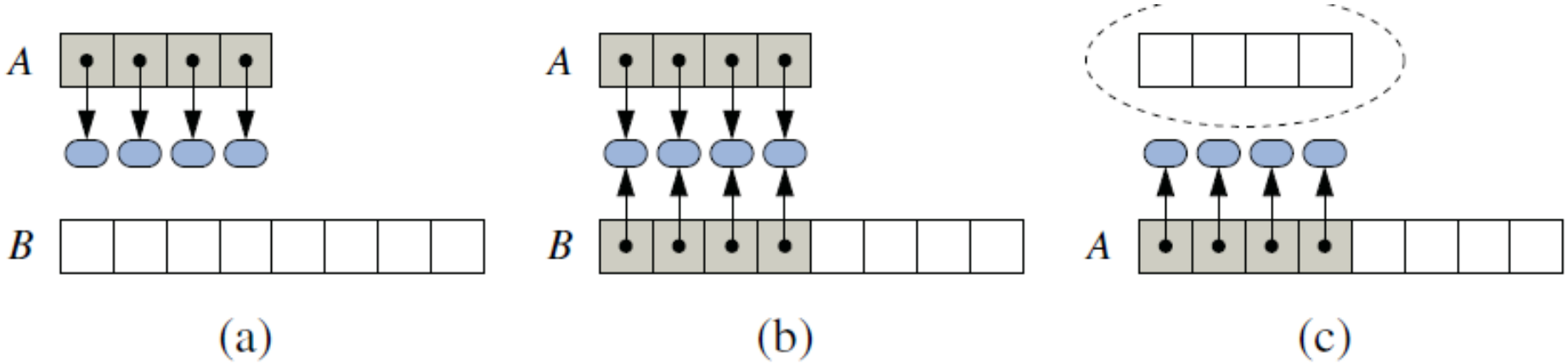
```
28    /** Inserts element e to be at index i, shifting all subsequent elements later. */
29    public void add(int i, E e) throws IndexOutOfBoundsException,
30                                                    IllegalStateException {
31      checkIndex(i, size + 1);
32      if (size == data.length)                      // not enough capacity
33        throw new IllegalStateException("Array is full");
34      for (int k=size-1; k >= i; k--)               // start by shifting rightmost
35        data[k+1] = data[k];
36      data[i] = e;                                  // ready to place the new element
37      size++;
38    }
39    /** Removes/returns the element at index i, shifting subsequent elements earlier. */
40    public E remove(int i) throws IndexOutOfBoundsException {
41      checkIndex(i, size);
42      E temp = data[i];
43      for (int k=i; k < size-1; k++)                // shift elements to fill hole
44        data[k] = data[k+1];
45      data[size-1] = null;                          // help garbage collection
46      size--;
47      return temp;
48    }
49    // utility method
50    /** Checks whether the given index is in the range [0, n-1]. */
51    protected void checkIndex(int i, int n) throws IndexOutOfBoundsException {
52      if (i < 0 || i >= n)
53        throw new IndexOutOfBoundsException("Illegal index: " + i);
54    }
55  }
```

Growable *(expandible)* array list basada en arreglo

- Dada la operación ***push(o)*** que agrega un elemento al final de la lista :
- Cuando el arreglo está lleno, lo reemplazamos por uno más largo
- ¿Cuanto de largo?
- Estrategia Incremental: se incrementa un tamaño constante ***c***
- Estrategia de duplicación: el doble del tamaño

```
Algorithm push(o)
  if t = S.length - 1 then
    A ← new array of
          size …
    for i ← 0 to n−1 do
      A[i] ← S[i]
    S ← A
  n ← n + 1
  S[n−1] ← o
```

Growable *(expandible)* array list basada en arreglo



Comparación de estrategias

- Comparemos la estrategia incremental y la de duplicación analizando el tiempo total ***T***(***n***) necesario para ejecutar una serie de ***n*** operaciones ***push***
- De esta manera, hay muchas operaciones de inserción simples para cada una de las costosas (las de duplicar el arreglo). Este hecho nos permite mostrar que una serie de operaciones ***push*** en un arreglo dinámico inicialmente vacío es eficiente en términos de su tiempo total de ejecución.
- Asumimos que iniciamos con una lista vacía representado por un arreglo creciente de tamaño 1
- Llamamos ***tiempo amortizado*** de una operación ***push***, al promedio de tiempo que toma una operación ***push*** sobre la serie de operaciones, ej., ***T***(***n***)/***n***

Análisis de la estrategia incremental

- Sobre las ***n*** operaciones ***push***, reemplazamos el arreglo ***k*** = ***n/c*** , donde ***c*** es una constante
- El tiempo total ***T***(***n***) de una serie de ***n*** operaciones ***push***

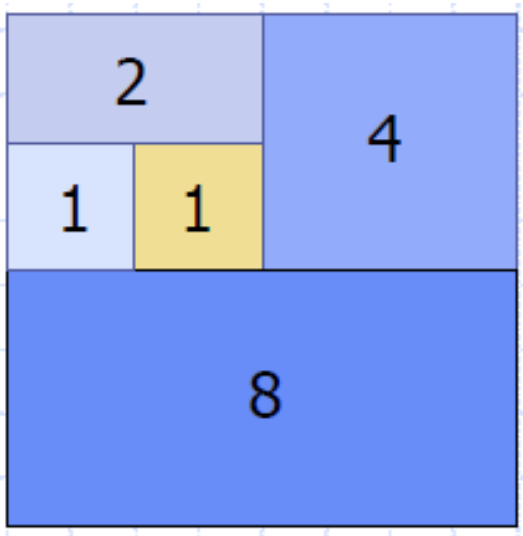
$$n + c + 2c + 3c + 4c + \ldots + kc =$$
$$n + c(1 + 2 + 3 + \ldots + k) =$$
$$n + ck(k + 1)/2$$

- Si ***c*** es una constante, ***T***(***n***) es ***O***(***n*** + ***k***^{2}),
- así, el tiempo amortizado de una operación push es ***O***(***n***)

Análisis de la estrategia de doblar

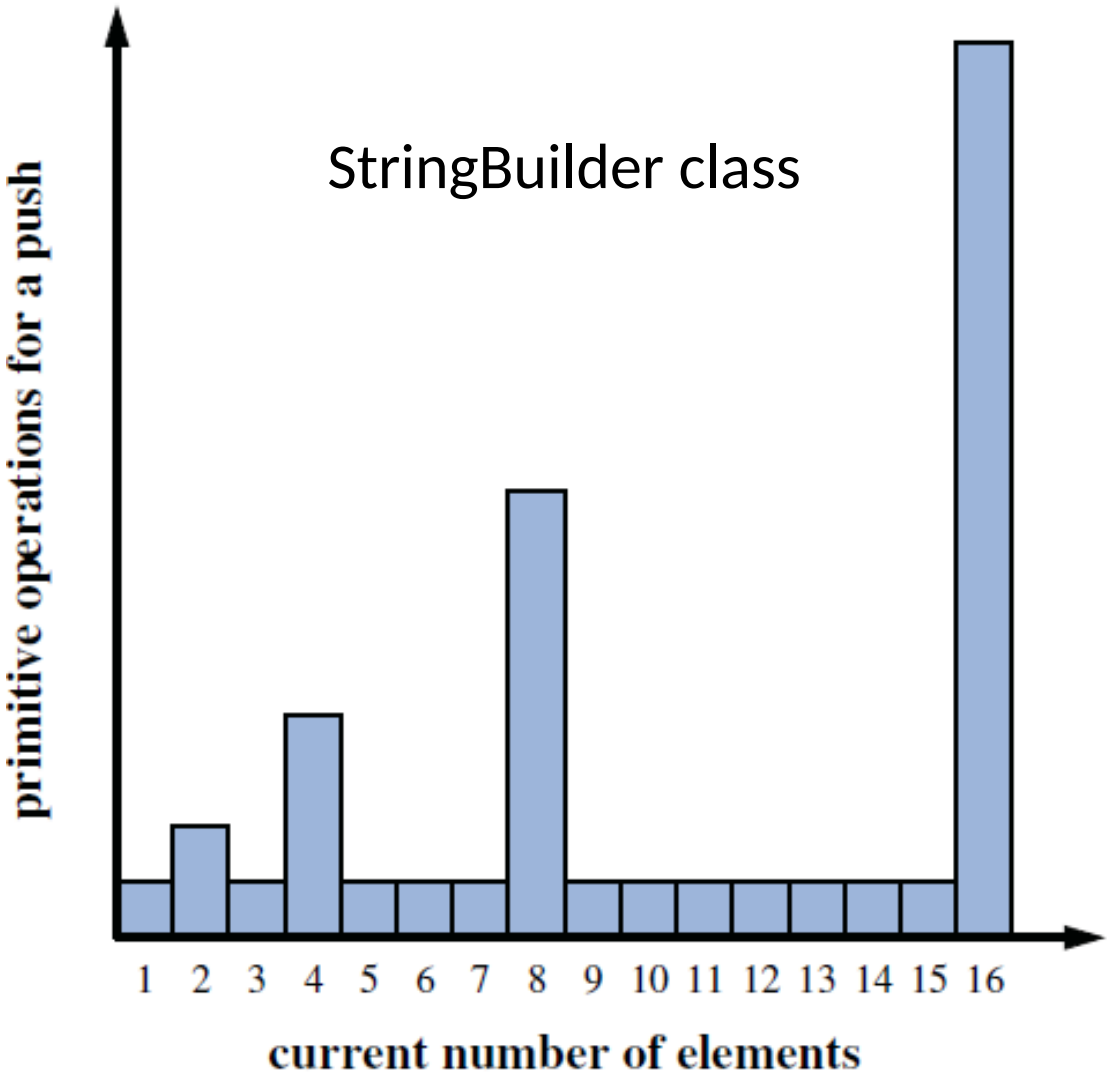
- El tiempo total ***T*(*n*)** de una serie ***n*** operaciones push es proporcional a:

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k$$



- ***T*(*n*)** es ***O*(*n*)**
- El tiempo amortizado de cada operación ***push*** es ***O*(1)**

Análisis de la estrategia de doblar



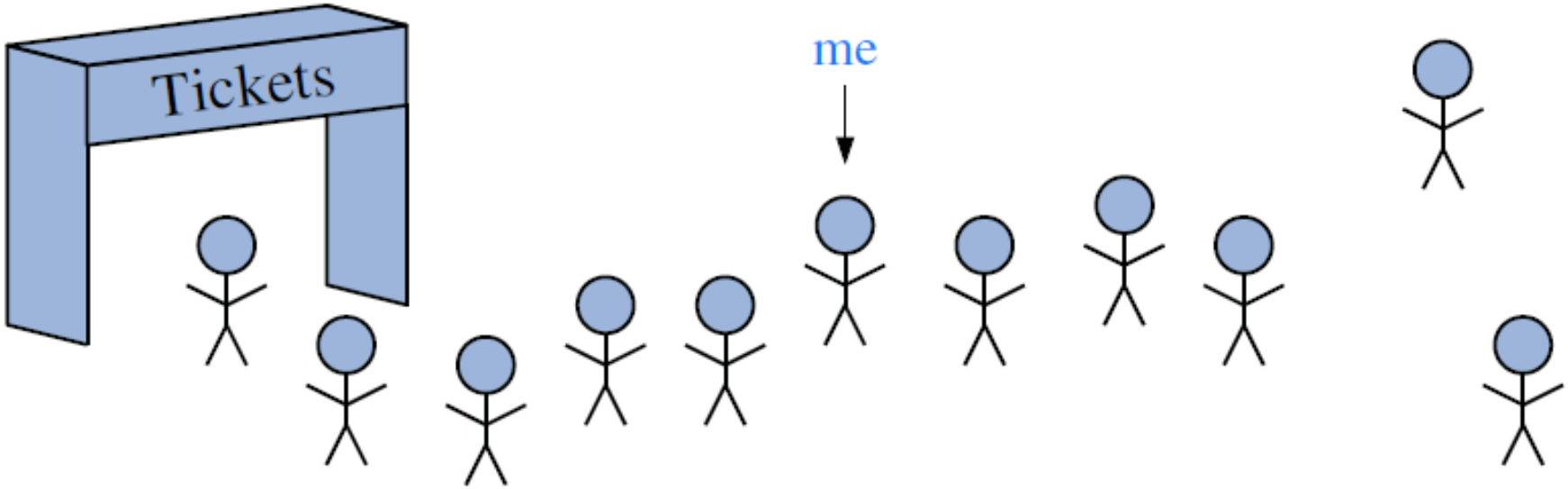
Positional List

- Un documento de texto puede verse como una larga secuencia de caracteres.
- Un procesador de texto utiliza la abstracción de un **cursor** para describir una posición.
- **Sin el uso explícito de un índice entero**, lo que permite operaciones como "eliminar el carácter en el cursor" o "insertar un nuevo carácter justo después del cursor".
- Además, es posible que podamos referirnos a una **posición inherente** dentro de un documento, como el comienzo de un capítulo en particular, sin depender de un índice de caracteres (o incluso un número de capítulo) que puede cambiar a medida que evoluciona el documento.

Positional List

- TAD lista de posiciones: proporciona una abstracción general de una secuencia de elementos con la capacidad de identificar la ubicación de un elemento.
- Una posición actúa como un marcador o token dentro de una lista de posición más amplia.
- Una posición ***p*** no se ve afectada por cambios en otra parte de una lista; la única manera en que una posición se convierte en ***no válida*** es si se emite un comando explícito para eliminarlo.
- Una instancia de posición es un objeto simple, que sólo admite el siguiente método:
  - ***P.getElement*** (): Devuelve el elemento almacenado en la posición **p**.

Positional List



TAD Positional List

- Métodos de acceso:

first( ): Returns the position of the first element of L (or null if empty).

last( ): Returns the position of the last element of L (or null if empty).

before(p): Returns the position of L immediately before position p (or null if p is the first position).

after(p): Returns the position of L immediately after position p (or null if p is the last position).

isEmpty( ): Returns true if list L does not contain any elements.

size( ): Returns the number of elements in list L.

Recorrido de Positional List

```
1  Position<String> cursor = guests.first( );
2  while (cursor != null) {
3    System.out.println(cursor.getElement( ));
4    cursor = guests.after(cursor);
5  }
```

Avanza al siguiente elemento de la lista

TAD Positional List

- Métodos de actualización:

addFirst(e): Inserts a new element e at the front of the list, returning the position of the new element.

addLast(e): Inserts a new element e at the back of the list, returning the position of the new element.

addBefore(p, e): Inserts a new element e in the list, just before position p, returning the position of the new element.

addAfter(p, e): Inserts a new element e in the list, just after position p, returning the position of the new element.

set(p, e): Replaces the element at position p with element e, returning the element formerly at position p.

remove(p): Removes and returns the element at position p in the list, invalidating the position.

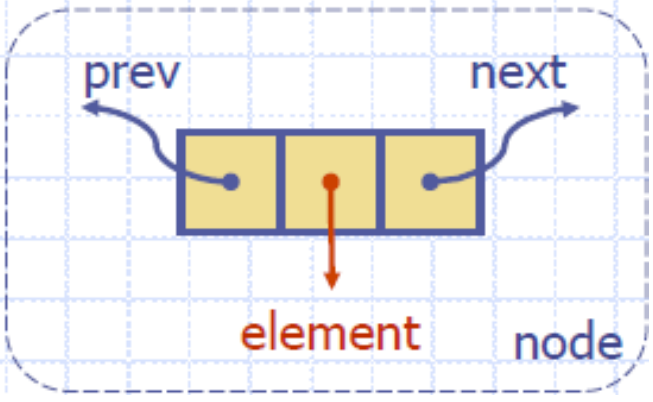
Ejemplo

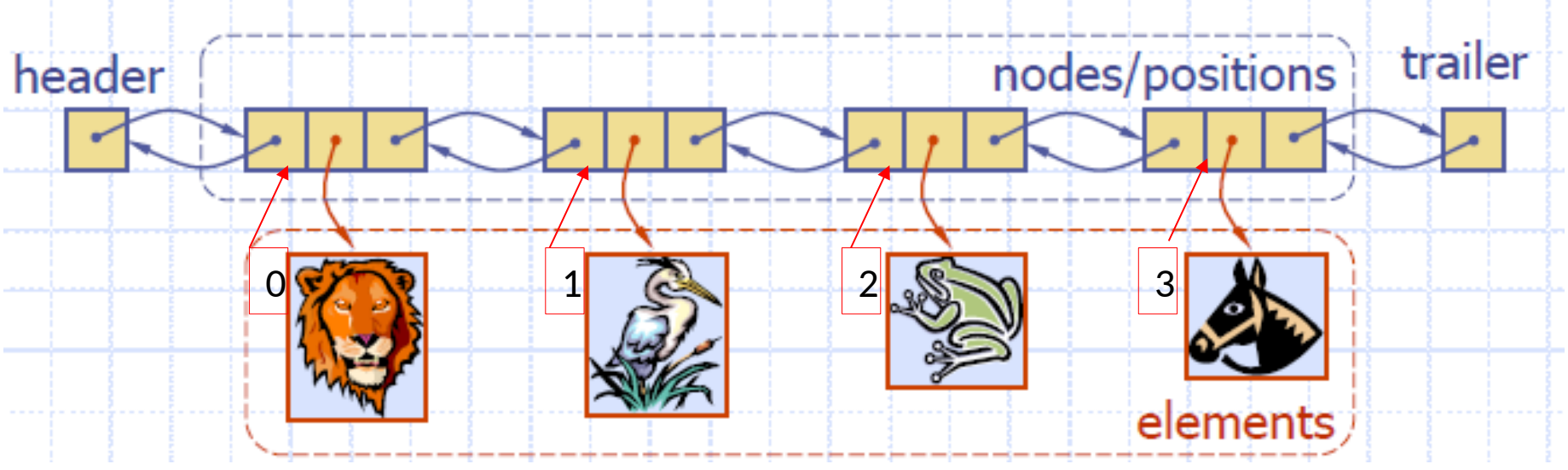
Method	Return Value	List Contents
addLast(8)	p	(8_p)
first( )	p	(8_p)
addAfter(p, 5)	q	$(8_p, 5_q)$
before(q)	p	$(8_p, 5_q)$
addBefore(q, 3)	r	$(8_p, 3_r, 5_q)$
r.getElement( )	3	$(8_p, 3_r, 5_q)$
after(p)	r	$(8_p, 3_r, 5_q)$
before(p)	null	$(8_p, 3_r, 5_q)$
addFirst(9)	s	$(9_s, 8_p, 3_r, 5_q)$
remove(last( ))	5	$(9_s, 8_p, 3_r)$
set(p, 7)	8	$(9_s, 7_p, 3_r)$
remove(q)	"error"	$(9_s, 7_p, 3_r)$

(Java *Position interface*)

Implementación de Positional List

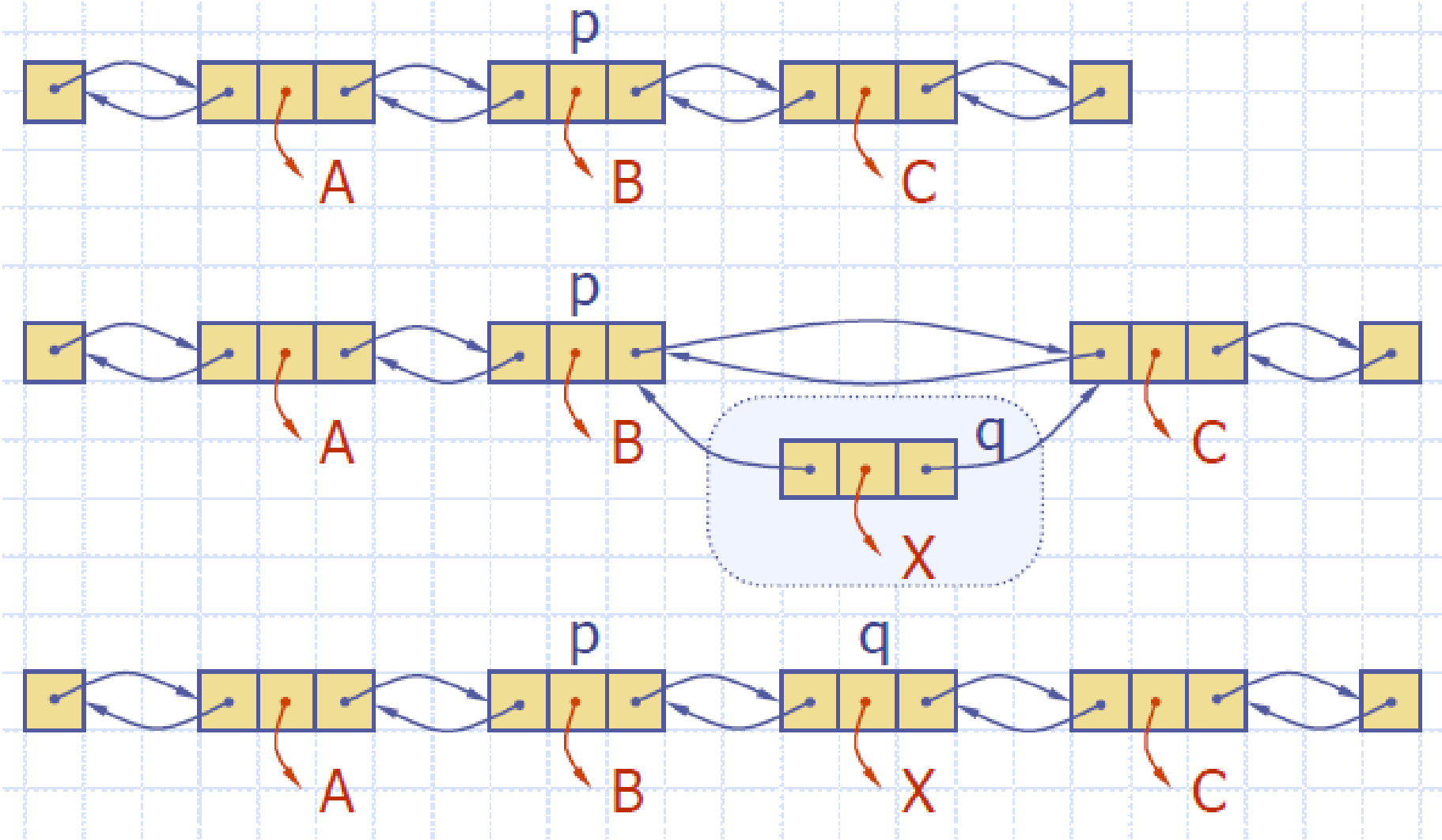
- La forma más natural es con una lista doblemente enlazada





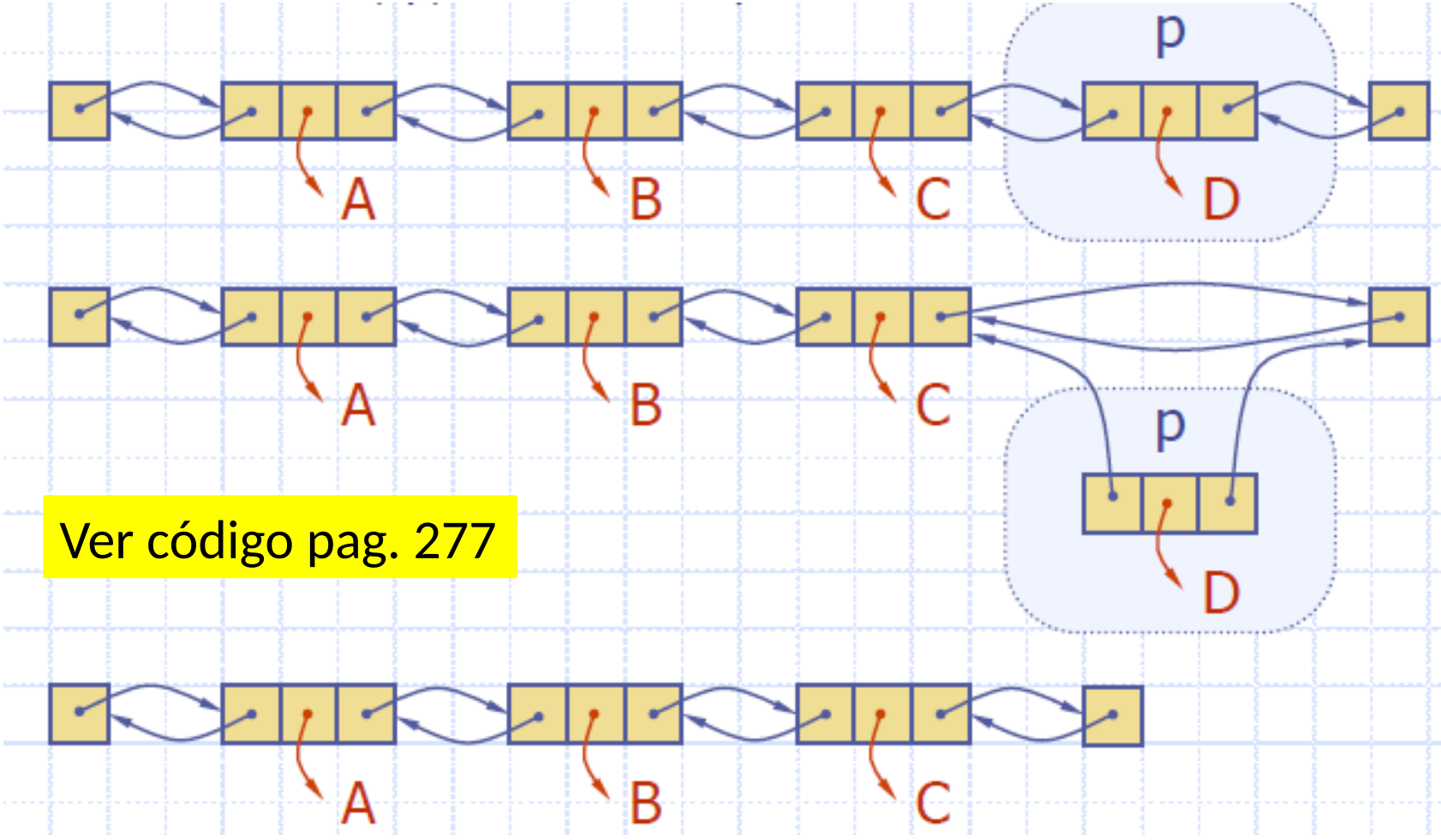
Inserción

- Inserta un nodo nuevo, ***q***, entre ***p*** y su ***sucesor***



Eliminación

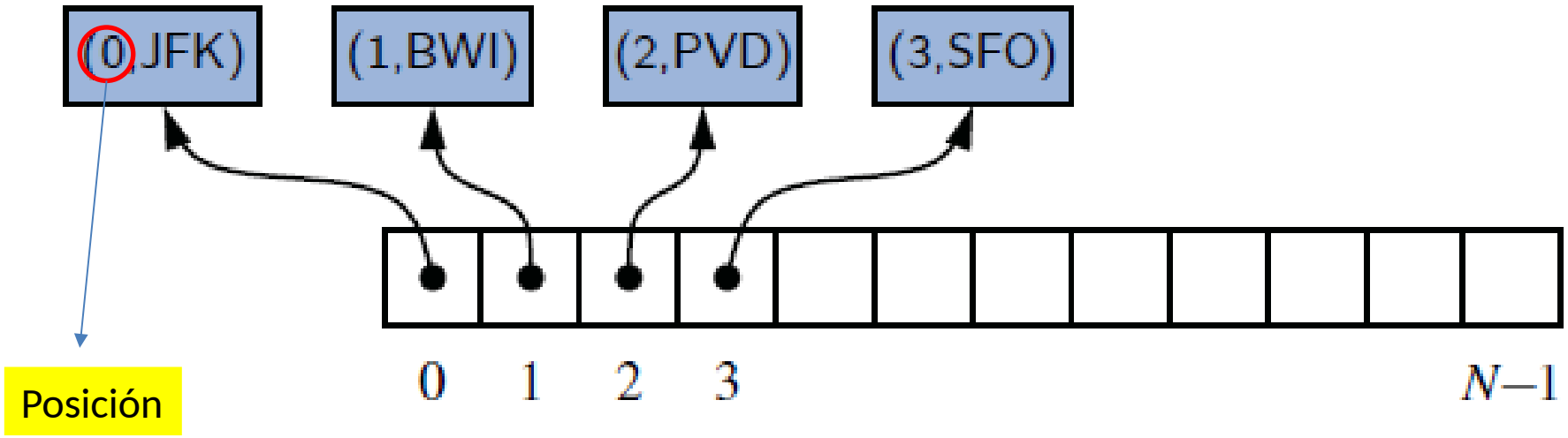
- Elimina un nodo, p, de la lista doble enlazada



Ver código pag. 277

Implementación Positional List

- Implementación con arreglo



Recuerde que las posiciones en una ***lista posicional*** siempre deben definirse en relación con sus posiciones vecinas, no con sus índices

Iterators (iteradores)

- Un ***iterador*** es un patrón de diseño de software que abstrae el proceso de recorrido (scanning), de a un elemento a la vez, a través de una secuencia de elementos.
- java.util.Iterator está definida la interfaz:

hasNext( ): Returns true if there is at least one additional element in the sequence, and false otherwise.

next( ): Returns the next element in the sequence.

```
while (iter.hasNext( )) {
  String value = iter.next( );
  System.out.println(value);
}
```

Iterators (iteradores)

- Es soportado por algunos *iteradores* el método:

remove( ): Removes from the collection the element returned by the most recent call to next( ). Throws an IllegalStateException if next has not yet been called, or if remove was already called since the most recent call to next.

- Si no se admite la eliminación, se genera una excepción ***UnsupportedOperationException***.
- En este curso trataremos de no usar este método

La interfaz Iterable

- Java define una interfaz parametrizada, denominada ***Iterable***, que incluye el siguiente método único:
- ***Iterator()***: Devuelve un iterador de los elementos de la colección.
- Una instancia de una clase de colección típica en Java, como ***ArrayList*,** es iterable (¡pero no es un iterador!); Produce un *iterador* para su colección como valor de retorno al invocar el método ***iterator()***.
- Cada llamada a ***iterator()*** devuelve una nueva instancia de *iterador*, permitiendo así recorridos múltiples (incluso simultáneos) de una colección.

El loop “for-each”

- La clase Java Iterable también juega un papel fundamental en el apoyo de la sintaxis del bucle "for-each":

```
for (ElementType variable : collection) {
  loopBody                                  // may refer to "variable"
}
```

```
ArrayList<Double> data;   // populate with random numbers (not shown)
Iterator<Double> walk = data.iterator( );
while (walk.hasNext( ))
  if (walk.next( ) < 0.0)
    walk.remove( );
```

Ver código iterations con clase ArrayList

El loop “for-each”

```
for (ElementType variable : collection) {
  loopBody                                    // may refer to "variable"
}
```

```
Iterator<ElementType> iter = collection.iterator( );
while (iter.hasNext( )) {
        ElementType variable = iter.next( );
        loopBody // may refer to ”variable”
}
```

Implementando iteradores

- ***snapshot (instantáneo) iterator*** mantiene su propia copia privada de la secuencia de elementos. O(n)
- ***lazy (perezoso) iterator*** es aquel que no realiza una copia inicial, sino que la realiza solo cuando se llama al método ***next()*** para solicitar otro elemento. O(1)

Ejemplo lazy

- Implementamos ***iteradores perezosos*** para ambas: ArrayList LinkedPositionalList, incluyendo el soporte para la operación de eliminación (pero sin ninguna garantía de falla rápida).

Iteraciones con la clase ArrayList

```
1   //---------------- nested ArrayIterator class ----------------
2   /**
3    * A (nonstatic) inner class. Note well that each instance contains an implicit
4    * reference to the containing list, allowing it to access the list's members.
5    */
6   private class ArrayIterator implements Iterator<E> {
7     private int j = 0;                          // index of the next element to report
8     private boolean removable = false;          // can remove be called at this time?
9
10    /**
11     * Tests whether the iterator has a next object.
12     * @return true if there are further objects, false otherwise
13     */
14    public boolean hasNext( ) { return j < size; }  // size is field of outer instance
15
```

Iteraciones con la clase ArrayList

```
16      /**
17       * Returns the next object in the iterator.
18       *
19       * @return next object
20       * @throws NoSuchElementException if there are no further elements
21       */
22      public E next( ) throws NoSuchElementException {
23        if (j == size) throw new NoSuchElementException("No next element");
24        removable = true;     // this element can subsequently be removed
25        return data[j++];     // post-increment j, so it is ready for future call to next
26      }
27
```

Iteraciones con la clase ArrayList

```
28      /**
29       * Removes the element returned by most recent call to next.
30       * @throws IllegalStateException if next has not yet been called
31       * @throws IllegalStateException if remove was already called since recent next
32       */
33      public void remove( ) throws IllegalStateException {
34        if (!removable) throw new IllegalStateException("nothing to remove");
35        ArrayList.this.remove(j−1);      // that was the last one returned
36        j−−;                             // next element has shifted one cell to the left
37        removable = false;               // do not allow remove again until next is called
38      }
39    } //----------- end of nested ArrayIterator class -----------
40
41    /** Returns an iterator of the elements stored in the list. */
42    public Iterator<E> iterator( ) {
43      return new ArrayIterator( );       // create a new instance of the inner class
44    }
```

Iteraciones con la clase LinkedPositionalList

```
1   //---------------- nested PositionIterator class ----------------
2   private class PositionIterator implements Iterator<Position<E>> {
3     private Position<E> cursor = first( );     // position of the next element to report
4     private Position<E> recent = null;         // position of last reported element
5     /** Tests whether the iterator has a next object. */
6     public boolean hasNext( ) { return (cursor != null);     }
7     /** Returns the next position in the iterator. */
8     public Position<E> next( ) throws NoSuchElementException {
9       if (cursor == null) throw new NoSuchElementException("nothing left");
10      recent = cursor;                  // element at this position might later be removed
11      cursor = after(cursor);
12      return recent;
13    }
```

Iteraciones con la clase LinkedPositionalList

```
14      /** Removes the element returned by most recent call to next. */
15      public void remove( ) throws IllegalStateException {
16        if (recent == null) throw new IllegalStateException("nothing to remove");
17        LinkedPositionalList.this.remove(recent);                  // remove from outer list
18        recent = null;                          // do not allow remove again until next is called
19      }
20    } //------------ end of nested PositionIterator class ------------
21
22    //---------------- nested PositionIterable class ----------------
23    private class PositionIterable implements Iterable<Position<E>> {
24      public Iterator<Position<E>> iterator( ) { return new PositionIterator( ); }
25    } //------------ end of nested PositionIterable class ------------
26
```

Iteraciones con la clase LinkedPositionalList

```
27    /** Returns an iterable representation of the list's positions. */
28    public Iterable<Position<E>> positions( ) {
29      return new PositionIterable( );          // create a new instance of the inner class
30    }
31
32    //---------------- nested ElementIterator class ----------------
33    /* This class adapts the iteration produced by positions() to return elements. */
34    private class ElementIterator implements Iterator<E> {
35      Iterator<Position<E>> posIterator = new PositionIterator( );
36      public boolean hasNext( ) { return posIterator.hasNext( ); }
37      public E next( ) { return posIterator.next( ).getElement( ); } // return element!
38      public void remove( ) { posIterator.remove( ); }
39    }
40
41    /** Returns an iterator of the elements stored in the list. */
42    public Iterator<E> iterator( ) { return new ElementIterator( ); }
```

Análisis Complejidad

Positional List ADT Method	java.util.List Method	ListIterator Method	Notes
size( )	size( )		$O(1)$ time
isEmpty( )	isEmpty( )		$O(1)$ time
	get(i)		A is $O(1)$, L is $O(\min\{i, n-i\})$
first( )	listIterator( )		first element is next
last( )	listIterator(size( ))		last element is previous
before(p)		previous( )	$O(1)$ time
after(p)		next( )	$O(1)$ time
set(p, e)		set(e)	$O(1)$ time
	set(i, e)		A is $O(1)$, L is $O(\min\{i, n-i\})$
	add(i, e)		$O(n)$ time
addFirst(e)	add(0, e)		A is $O(n)$, L is $O(1)$
addFirst(e)	addFirst(e)		only exists in L, $O(1)$
addLast(e)	add(e)		$O(1)$ time
addLast(e)	addLast(e)		only exists in L, $O(1)$
addAfter(p, e)		add(e)	insertion is at cursor; A is $O(n)$, L is $O(1)$
addBefore(p, e)		add(e)	insertion is at cursor; A is $O(n)$, L is $O(1)$
remove(p)		remove( )	deletion is at cursor; A is $O(n)$, L is $O(1)$
	remove(i)		A is $O(1)$, L is $O(\min\{i, n-i\})$

Framework de colecciones de Java

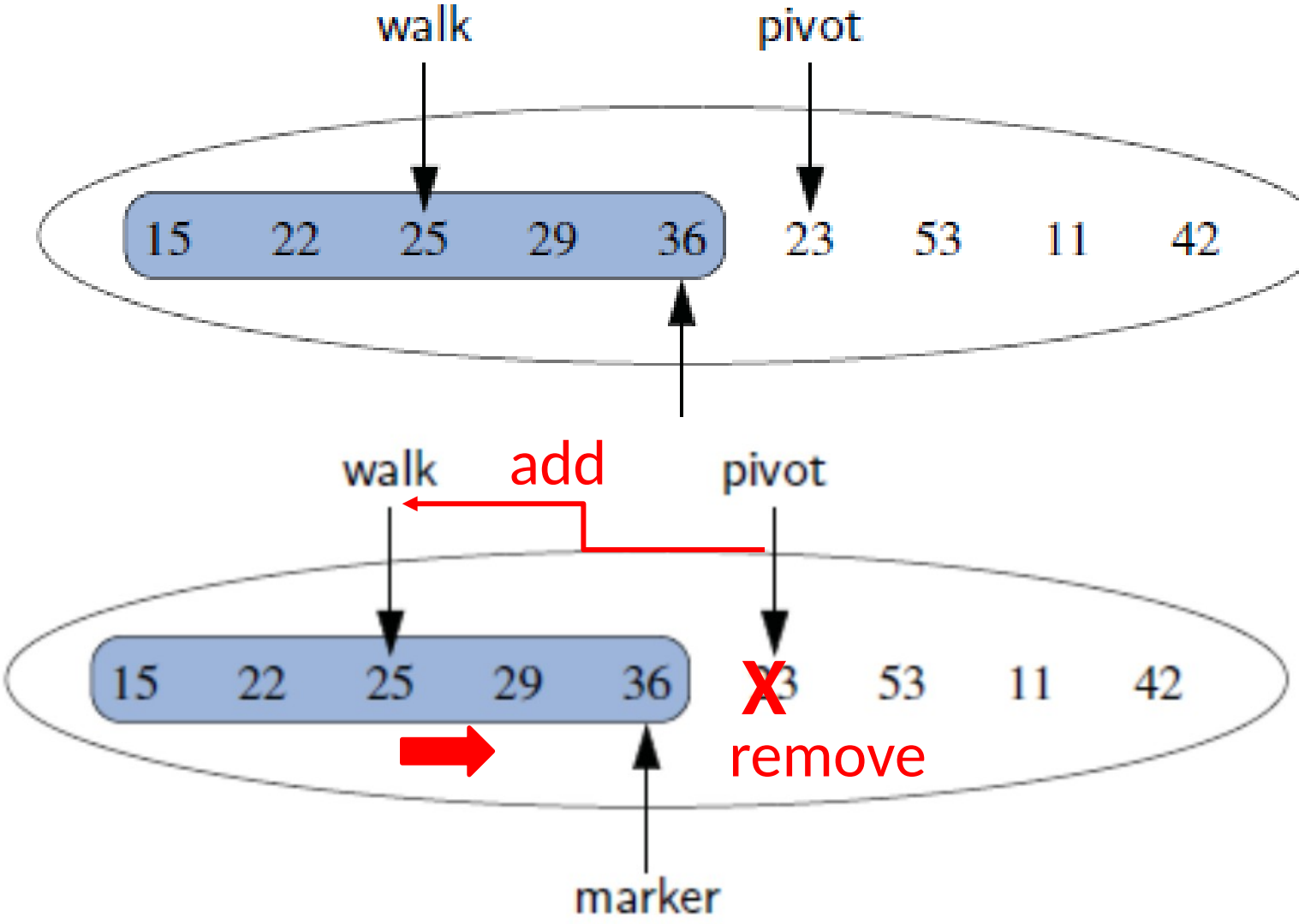
Class	Interfaces			Properties			Storage	
	Queue	Deque	List	Capacity Limit	Thread-Safe	Blocking	Array	Linked List
ArrayBlockingQueue	✓			✓	✓	✓	✓	
LinkedBlockingQueue	✓			✓	✓	✓		✓
ConcurrentLinkedQueue	✓				✓		✓	
ArrayDeque	✓	✓					✓	
LinkedBlockingDeque	✓	✓		✓	✓	✓		✓
ConcurrentLinkedDeque	✓	✓			✓			✓
ArrayList			✓				✓	
LinkedList	✓	✓	✓					✓

Algoritmos basados en listas en el framework de colecciones de Java

- Conversión de Listas en Arreglos
  - **toArray( ):** retorna un arreglo de elementos del tipo de objetos que contiene la colección.
  - **asArray(*A*):** retorna un arreglo de elementos del mismo tipo que contiene la colección A, conteniendo todos los elementos de ella.
- Conversión de Arreglos en Listas
  - **asList(*A*)**: retorna una lista que representa el arreglo A, con los mismos tipos de elementos que A.

```
Integer[ ] arr = {1, 2, 3, 4, 5, 6, 7, 8};
List<Integer> listArr = Arrays.asList(arr);
```

Inserción ordenada en lista posicional



Inserción ordenada en lista posicional

```
1  /** Insertion-sort of a positional list of integers into nondecreasing order */
2  public static void insertionSort(PositionalList<Integer> list) {
3    Position<Integer> marker = list.first( );      // last position known to be sorted
4    while (marker != list.last( )) {
5      Position<Integer> pivot = list.after(marker);
6      int value = pivot.getElement( );            // number to be placed
7      if (value > marker.getElement( ))           // pivot is already sorted
8        marker = pivot;
9      else {                                      // must relocate pivot
10       Position<Integer> walk = marker;          // find leftmost item greater than value
11       while (walk != list.first( ) && list.before(walk).getElement( ) > value)
12         walk = list.before(walk);
13       list.remove(pivot);                       // remove pivot entry and
14       list.addBefore(walk, value);              // reinsert value in front of walk
15     }
16   }
17 }
```

Bibliografía

- Data Structures and Algorithms in Java™. Sixth Edition. Michael T. Goodrich, Department of Computer Science University of California. Roberto Tamassia, Department of Computer Science Brown University. Michael H. Goldwasser, Department of Mathematics and Computer Science Saint Louis University. Wiley. 2014.
- https://www.geeksforgeeks.org/