Natividad Prieto
Francisco Marqués
Isabel Galiano
Jorge González
Carlos Martínez-Hinarejos
Javier Piris
Assumpció Casanova
Marisa Llorens
Jon Ander Gómez
Carlos Herrero
Germán Moltó

Capítulo 16

Tipos lineales. Estructuras enlazadas

Los tipos de datos lineales son aquéllos cuyos elementos están formados por linealidades o secuencias

$$d_0 d_1 ... d_{n-1}, \quad n \geq 0$$

en los que todos los d_i son datos del mismo tipo y sobre los que, en términos generales, se pueden hacer operaciones de inserción, búsqueda, eliminación de datos, consulta del dato que ocupa una determinada posición, etc.

Los diversos tipos de datos lineales surgen de aquellas clases de problemas en los que se manejan secuencias de datos y que necesitan una política concreta de gestión de sus elementos. Por ejemplo, en una secuencia de elementos puestos en cola para ser tratados o atendidos por orden de llegada, no se pueden permitir las mismas operaciones o métodos que sobre una lista de elementos entre los que no exista dicha restricción en su tratamiento.

En este capítulo se van a presentar tres tipos lineales: `Pila`, `Cola` y `Lista`. Estas clases se consideran básicas por ser idóneas en una gran variedad de aplicaciones informáticas, por lo que su funcionalidad e implementación está ampliamente estudiada. Por ejemplo, como se vio en el capítulo 4, Java gestiona la lista de registros de activación de las llamadas pendientes de terminar como una *pila*: todos los registros permanecen inaccesibles salvo el de la cima. En cambio, la lista de peticiones de compra electrónica de entradas se gestiona habitualmente como una *cola*.

Como se verá a lo largo del capítulo, el catálogo de métodos requerido por cada tipo lineal influye decisivamente en la forma más adecuada y eficiente de estructurar los datos. De hecho, en lugar de la representación mediante arrays que se discutió en la sección 9.4, en algunos casos será especialmente indicado el uso de las *listas* o *secuencias enlazadas* que se introducen en la siguiente sección.

16.1 Representación enlazada de secuencias

Una manera obvia de representar una secuencia consiste en usar un array, declarado de longitud suficientemente grande, en el que disponer consecutivamente los sucesivos elementos de la secuencia. Esta representación permite acceder directamente a cualquier elemento, independientemente de la posición que ocupe en la secuencia, pero resulta muy ineficiente cuando se realizan muchos movimientos de datos al añadir o eliminar elementos en posiciones intermedias.

16.1.1 Definición recursiva de secuencias. La clase `Nodo`

Una representación alternativa a la anterior se basa en disponer de memoria para los datos a medida que se van insertando en la secuencia, de modo que a diferencia de lo que sucede con las componentes de un array:

- los elementos consecutivos en la secuencia no tienen por qué aparecer consecutivos en memoria, y
- en la declaración de una secuencia no aparece limitado el número de elementos que pueden formar parte de ella.

Esta representación se basa en que todo dato tiene asociado un *enlace* o referencia a la posición en que se encuentra en el heap el siguiente dato de la secuencia, como se muestra en la figura 16.1.

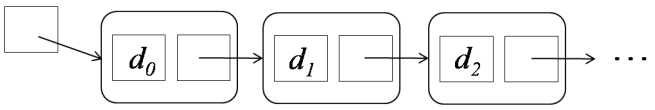


Figura 16.1: Representación enlazada de la secuencia $d_0 d_1 d_2 \ldots$.

Genéricamente, se denomina *nodo* al objeto que agrupa un `dato` d de un tipo `T` cualquiera y un enlace `siguiente`, como se representa gráficamente en la figura 16.2.

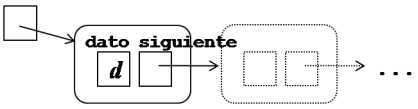


Figura 16.2: Estructura de un nodo.

Los nodos ayudan a definir recursivamente las secuencias enlazadas en términos de ellas mismas. Así pues, como puede verse en la figura 16.3, una *secuencia enlazada* de un número cualquiera $n \geq 0$ de datos es:

- la secuencia de $n = 0$ datos, en cuyo caso vale `null`, o
- una secuencia enlazada de $n \geq 1$ datos, en cuyo caso es un objeto nodo con un dato seguido por una secuencia enlazada de $n - 1$ datos.

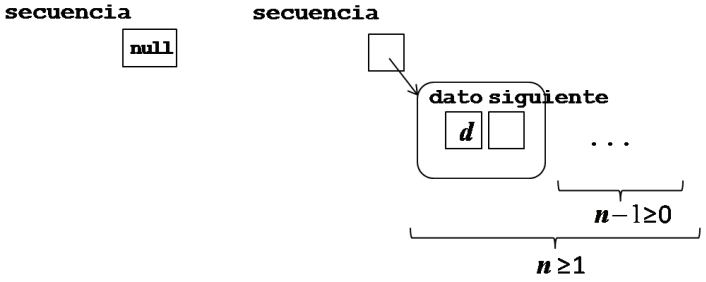


Figura 16.3: Secuencia enlazada de 0 o más datos.

En el caso particular en que el dato es de tipo `int` la clase `NodoInt` es la que se muestra en la figura 16.4.

De los atributos de la clase cabe resaltar:

- El atributo `siguiente` se declara de la propia clase `NodoInt`; Java lo permite, debiéndose entender como una definición recursiva.
- Los atributos `dato` y `siguiente` se han declarado *friendly* para que algunas clases, como las de la sección 16.2, puedan acceder a ellos y manejarlos en la implementación de sus operaciones.

La clase se completa con dos métodos constructores:

- `NodoInt(int)`, que crea un nodo sin ningún otro a continuación.
- `NodoInt(int, NodoInt)`, que permite crear un nodo que antepone un dato entero a otro nodo previamente creado.

```
/**
 *  Clase NodoInt: representa un nodo de una secuencia enlazada
 *  que tiene un dato de tipo int y un enlace al nodo siguiente.
 *  @author Libro IIP-PRG
 *  @version 2016
 */
class NodoInt {
    int dato;
    NodoInt siguiente;

    /** Crea un nodo con un dato d y sin siguiente.
     *  @param d int, el dato del nuevo nodo.
     */
    NodoInt(int d) {
        dato = d;
        siguiente = null;
    }

    /** Crea un nodo con un dato d, enlazado a un nodo
     *  ya existente s.
     *  @param d int, el dato del nuevo nodo.
     *  @param s NodoInt, con el que enlazar el nuevo nodo.
     */
    NodoInt(int d, NodoInt s) {
        dato = d;
        siguiente = s;
    }
}
```

Figura 16.4: Clase `NodoInt`

Las figuras 16.5(a) y 16.5(b) muestran el resultado de asignar a una variable `NodoInt sec` el nodo `new NodoInt(d)` y el nodo `new NodoInt(d, s)`, respectivamente.

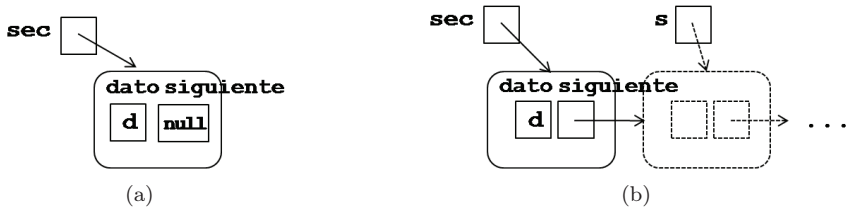


Figura 16.5: Los dos constructores de `NodoInt`.

Ejemplo 16.1. Supóngase una variable `sec` declarada de tipo `NodoInt` y la siguiente secuencia de instrucciones:

```
NodoInt sec = null;
sec = new NodoInt(10);
sec = new NodoInt(5, sec);
sec = new NodoInt(-2, sec);
```

La variable `sec` va pasando por los estados que se muestran en la figura 16.6. El número de sus datos va aumentando desde 0 hasta 3; dado que el constructor sitúa cada nuevo nodo a la cabeza de `sec`, los datos quedan en el orden `-2 5 10`.

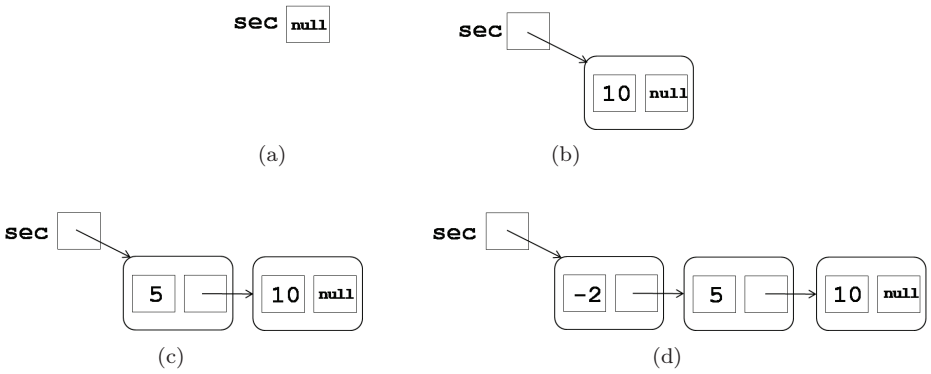


Figura 16.6: Formación de una secuencia enlazada de tres elementos.

El primer constructor de `NodoInt` es un caso particular del segundo y, de hecho, se podría haber definido como:

```
NodoInt(int d) {
    this(d, null);
}
```

Así, en el ejemplo 16.1 las primeras líneas de código se podrían haber escrito también, con idéntico resultado, como:

```
NodoInt sec = null;
sec = new NodoInt(10, sec);
```

Ejemplo 16.2. Dado un entero $n \geq 1$, el siguiente código crea la secuencia de los n primeros impares, desde 1 hasta $2n - 1$ inclusive:

```
NodoInt sec = null;
for (int i = 2 * n - 1; i >= 1; i -= 2) { sec = new Nodo(i, sec); }
```

En los ejemplos 16.1 y 16.2 se muestra cómo el segundo constructor `NodoInt(int, NodoInt)` facilita la inserción de un nuevo elemento en la cabeza de la secuencia, disponiendo de la memoria a medida que aumenta la talla de la secuencia. Pero el uso explícito de los enlaces permite acceder a otras posiciones de la secuencia, como en el ejemplo siguiente.

Ejemplo 16.3. El siguiente código se supone escrito en una clase con acceso *friendly* a los atributos de `NodoInt`. El primer nodo creado se sitúa como antes en cabeza de la secuencia `sec` y pasa a ser también el último de la secuencia. Las dos siguientes inserciones añaden el nuevo nodo a continuación de `ultimo` y después se actualiza en consecuencia dicha variable (figura 16.7):

```
NodoInt sec = null, ultimo = null;
sec = new NodoInt(10); ultimo = sec;
ultimo.siguiente = new NodoInt(5); ultimo = ultimo.siguiente;
ultimo.siguiente = new Nodo(-2); ultimo = ultimo.siguiente;
```

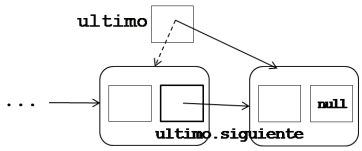


Figura 16.7: Uso explícito del enlace `siguiente` en una secuencia enlazada.

Así, la variable `sec` va pasando por los estados que se muestran en la figura 16.8. El número de sus datos va aumentando como en el ejemplo de la figura 16.6 desde 0 hasta 3, pero quedan en el orden `10 5 -2`.

La clase `NodoInt` se considera como una clase subsidiaria que, al igual que los arrays, ayuda a dar estructura a los componentes o atributos de diversos tipos de datos. La clase se ha declarado *friendly* dado que su cometido básico es dar soporte a los enlaces con que se materializan las secuencias enlazadas. Como ya se ha comentado, su repertorio de métodos se limita a sus dos constructores y los atributos se dejan *friendly* para que se puedan manipular explícitamente, como en el ejemplo 16.3, por las clases que incluyan a `NodoInt` en su paquete.

Las clases externas a `NodoInt` pueden precisar encapsular en un método el tratamiento de una secuencia enlazada. Hay que tener en cuenta que al ser la secuencia un parámetro del método, se deberá terminar devolviendo como resultado la propia secuencia si dicho parámetro sufre localmente algún cambio.

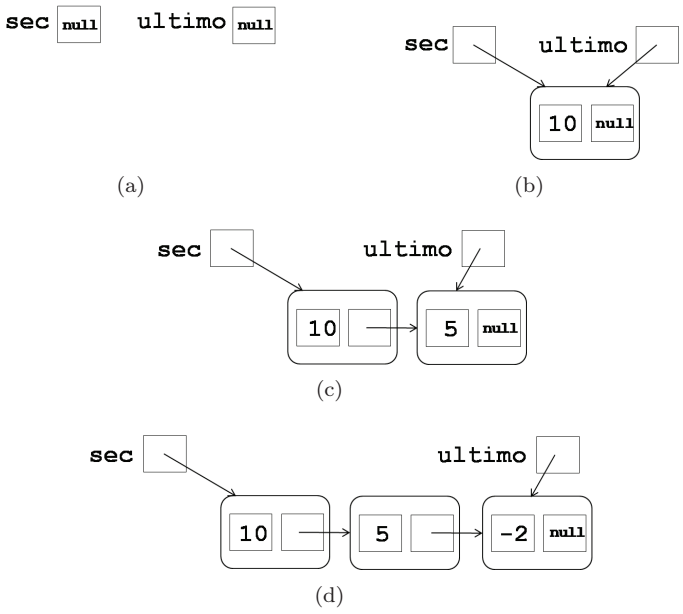


Figura 16.8: Formación de una secuencia enlazada de tres elementos. Los nuevos elementos se enlazan al final de la secuencia.

Ejemplo 16.4. Se tiene un programa que manipula secuencias enlazadas de enteros que pueden empezar por un código entre 90 y 95 inclusive. Se necesita actualizar estas secuencias para que los códigos se limiten a tres, según:

- Los tres primeros códigos (entre 90 y 92 inclusive) se cambiarán a 91 y los tres últimos (entre 93 y 95 inclusive) se cambiarán a 92.
- Las que no disponen de este código se hará que empiecen por 90.

Para ello se define el siguiente método:

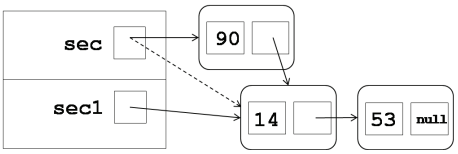
```
public static NodoInt norma90(NodoInt sec) {
    if (sec != null) {
        if (90 <= sec.dato && sec.dato <= 92) { sec.dato = 91; }
        else if (93 <= sec.dato && sec.dato <= 95) { sec.dato = 92; }
        else { sec = new NodoInt(90, sec); }
    }
    return sec;
}
```

El código anterior realiza en algunos casos cambios en nodos que se encuentran en el montículo, pero en un caso se modifica localmente el parámetro `sec`, por lo que

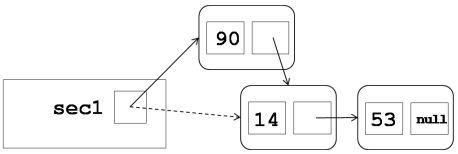
hay que devolver `sec` para que dicho cambio sobreviva a la llamada al método. La figura 16.9 ilustra este caso mostrando los cambios producidos por

```
sec1 = norma90(sec1);
```

en donde `sec1` es una secuencia iniciada con los datos `14 53`.



(a) Modificación de la variable local `sec`.



(b) Asignación a `sec1` del resultado de la llamada.

Figura 16.9: Actualización de `sec1` por el método `norma90`.

16.1.2 Recorrido y búsqueda en secuencias enlazadas

A diferencia de lo que sucede con los arrays, en las secuencias enlazadas no se tiene acceso directamente al elemento que ocupa una posición determinada en la secuencia, sino que sólo se puede acceder a un nodo desde su anterior. Es por ello que la mayoría de operaciones sobre estas secuencias requieren un recorrido o una búsqueda desde el primer nodo en adelante.

La estructura general de los algoritmos de recorrido y búsqueda en secuencias enlazadas es análoga a la de los esquemas de recorrido y búsqueda ascendentes en arrays (en sección 9.3.1 del capítulo 9). En un esquema iterativo se usa una variable que, como en el ejemplo anterior, indica en cada momento el primer elemento de la subsecuencia que todavía no se ha revisado (figura 16.10). Dicha variable:

- se sitúa inicialmente sobre el primer elemento,
- en cada pasada del bucle da acceso al dato a procesar y, después, se actualiza al siguiente elemento,
- en el estado final, el valor `null` de dicha variable indica que no queda ningún dato por procesar.

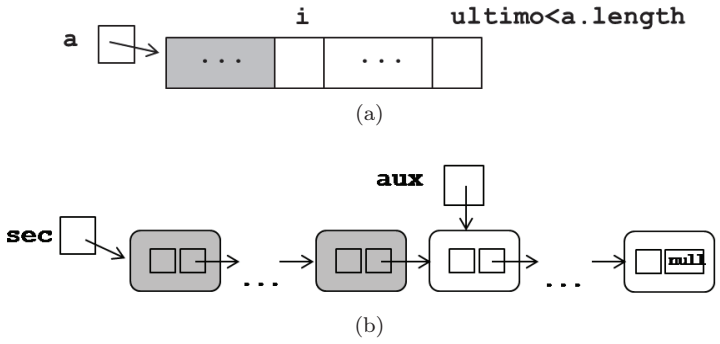


Figura 16.10: Recorridos de arrays y secuencias enlazadas.

En la discusión de los esquemas que vienen a continuación se suponen declarados un array `a` y un índice `ultimo` dentro del rango de `a`; por otra parte se supone `sec` de tipo `NodoInt`.

Esquemas de recorrido

Los esquemas generales de un *recorrido de un array y de una secuencia enlazada* se muestran a continuación:

```
int i = 0;
while (i <= ultimo) {
    tratar(a[i]);
    i++;
}
```

```
NodoInt aux = sec;
while (aux != null) {
    tratar(aux.dato);
    aux = aux.siguiente;
}
```

donde `tratar(a[i])` y `tratar(aux.dato)` indican, respectivamente, la operación a realizar con el elemento i-ésimo del array y con el dato del nodo `aux`. En ambos esquemas, el recorrido de n elementos es $\Theta(n)$.

Ejemplo 16.5. Se tiene una secuencia enlazada `sec` de enteros. Se desea que los valores de la lista saturen a un cierto valor `maximo`, es decir, que todos los valores >`maximo` se cambien a `maximo`.

```
NodoInt aux = sec;
while (aux != null) {
    if (aux.dato > maximo) { aux.dato = maximo; }
    aux = aux.siguiente;
}
```

Esquemas de búsqueda

La búsqueda en una secuencia enlazada es, como en los arrays, una variación de un recorrido. El esquema general de una *búsqueda en un array* de un elemento que cumpla una cierta propiedad, que se concreta en el método `propiedad(x)` que comprueba si el valor `x` cumple la propiedad enunciada, se muestra a continuación:

```
int i = 0;
while (i <= ultimo && !propiedad(a[i])) { i++; }
// Resolución de la búsqueda
if (i <= ultimo) ...  // a[i] cumple la propiedad
else ...              // ningún elemento cumple la propiedad
```

El esquema general de una *búsqueda en una secuencia enlazada* de un elemento que cumpla una propiedad, tratada igual que en caso anterior, es el siguiente:

```
NodoInt aux = sec;
while (aux != null && !propiedad(aux.dato)) {
    aux = aux.siguiente;
}
// Resolución de la búsqueda
if (aux != null) ...  // aux.dato cumple la propiedad
else ...            // ningún elemento cumple la propiedad
```

El uso del operador cortocircuitado `&&` en la guarda del bucle asegura que sólo se accede al dato después de comprobar que `aux != null`.

En ambos esquemas coinciden las cotas de complejidad $\Omega(1)$ y $\mathrm{O}(n)$, siendo la talla n la longitud de la secuencia.

Ejemplo 16.6. El siguiente código busca la posición de la primera aparición en la secuencia del dato `d`. Si no aparece, la posición toma el valor -1.

```
NodoInt aux = sec; int i = 0, pos;
while (aux != null && aux.dato != d) {
    aux = aux.siguiente; i++;
}
if (aux != null) { pos = i; }
else { pos = -1; }
```

Este algoritmo se podría aplicar casi directamente a la búsqueda de un dato `d` en secuencias cuyos nodos contuviesen objetos en lugar de datos enteros o de otro tipo primitivo. Bastaría con sustituir la comparación `!=` propia de los tipos primitivos por su correspondiente `!aux.dato.equals(d)`.

El esquema de búsqueda también es aplicable a resolver el acceso secuencial a una determinada posición, propio de las secuencias enlazadas.

Ejemplo 16.7. El acceso al `i`-ésimo elemento se logra buscando el nodo `i`-ésimo y accediendo a sus atributos sólo en el caso en que dicho nodo exista. En el siguiente algoritmo la posición del nodo `aux` se va registrando en la variable `k`:

```
NodoInt aux = sec; int k = 0;
while (aux != null && k < i) {
    aux = aux.siguiente;
    k++;
}
```

Si el bucle acaba con `aux != null` entonces necesariamente `k == i` y `aux` es el `i`-ésimo nodo. En caso contrario, dicho nodo no existe.

El coste de esta operación es $\Theta(\mathtt{i})$ si $\mathtt{i} < n$, siendo n el número de elementos. En caso contrario, este algoritmo recorre toda la secuencia con un coste $\Theta(n)$ antes de detectar que dicha posición no existe. En resumen, el coste es $\Theta(\mathit{mínimo}(\mathtt{i}, n))$.

En el caso concreto en que se desee acceder al último elemento de la secuencia, aquel que no tiene siguiente, es innecesario comprobar repetidamente que `aux` es diferente de `null`, ya que si se comprueba al inicio que existe al menos un nodo, la existencia de un último nodo está asegurada:

```
if (sec != null) {
    NodoInt aux = sec;
    while (aux.siguiente != null) {
        aux = aux.siguiente;
    }
    System.out.println("El último dato es " + aux.dato);
}
else { System.out.println("La secuencia está vacía"); }
```

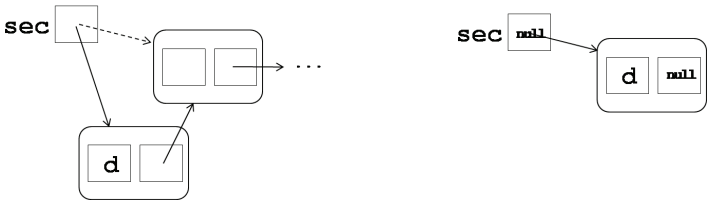
16.1.3 Inserción y borrado en secuencias enlazadas

Gracias al uso explícito de los enlaces entre nodos, las operaciones de inserción y borrado en cualquier posición de una secuencia se resuelven sin realizar ningún movimiento en memoria de los datos ya existentes en la secuencia. Para ello es conveniente utilizar una referencia hacia el nodo inmediatamente anterior a la posición en la que se va actuar, dado que en otro caso para la inserción es necesario desplazar al menos un dato y en el borrado hay posiciones en las que no es posible borrar o el proceso se complica bastante.

En primer lugar se va a considerar el problema de insertar un dato `d` en una secuencia `sec` dada. Se pueden dar dos casos según dónde se deba hacer la inserción:

- El nuevo nodo se inserta en la primera posición, lo que incluye el caso de insertar en una secuencia vacía (figuras 16.11(a) y 16.11(b)):

```
sec = new NodoInt(d, sec);
```



(a) (b)

Figura 16.11: Inserción en cabeza.

- Se inserta en cualquier otra ubicación diferente de `sec`. Ello supone que se debe insertar detrás de algún nodo, de modo que si `ant` es una referencia al nodo anterior a donde realizar la inserción:

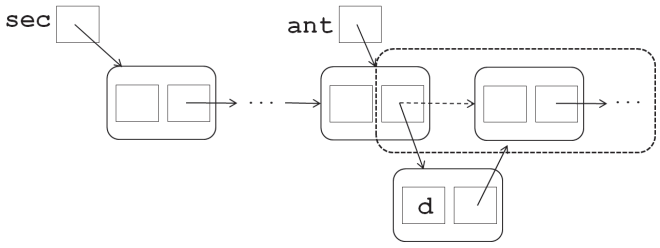
```
ant.siguiente = new NodoInt(d, ant.siguiente);
```

  inserta en cabeza de la subsecuencia `ant.siguiente`. Ello incluye el caso de insertar después del último nodo (figuras 16.12(a) y 16.12(b)).

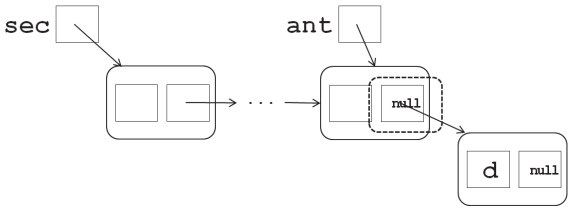
Ejemplo 16.8. Dada una secuencia enlazada con un cierto número n de nodos, se desea insertar el elemento `d` en la posición `i`, siempre que $0 \leq$ `i` $\leq n$.

El siguiente código comprueba si `i == 0`, en cuyo caso la inserción es en cabeza. Sino, busca el nodo que ocupa la posición `i - 1` para insertar a continuación un nuevo nodo con el dato `d`:

```
if (i == 0) { sec = new NodoInt(d, sec); }
else {
    NodoInt aux = sec; int k = 0;
    while (aux != null && k < i - 1) {
        aux = aux.siguiente;
        k++;
    }
    if (aux != null) {
        aux.siguiente = new NodoInt(d, aux.siguiente);
    }
}
```



(a)



(b)

Figura 16.12: Inserción detrás de un nodo.

Si la búsqueda termina con fracaso, el número de nodos de `sec` ha resultado ser menor que `i` y no se realiza ninguna inserción.

El coste $\Theta(\mathit{mínimo}(\texttt{i}, n))$ de la operación es debido a la búsqueda de la posición `i`-ésima, pues la inserción del nuevo nodo se resuelve con coste $\Theta(1)$ al no tener que realizar ningún movimiento de los datos preexistentes en la secuencia.

Aunque el tratamiento natural de las linealidades es en términos generales el iterativo, tiene interés considerar la versión recursiva. Para este ejemplo, se basa en el siguiente análisis de casos:

- Secuencia con $n = 0$ nodos, `sec == null`. Se inserta en cabeza de `sec`, si y solamente si `i == 0`.
- Secuencia con $n > 0$ nodos. Si `i == 0`, se inserta en cabeza de `sec`, sino el problema se reduce a insertar en la posición `i - 1` de la subsecuencia `sec.siguiente`.

Lo que da lugar al siguiente método:

```
public static NodoInt insertar(NodoInt sec, int d, int i) {
    if (sec == null) {
        if (i == 0) { sec = new NodoInt(d); }
    } else if (i == 0) { sec = new NodoInt(d, sec); }
    else { sec.siguiente = insertar(sec.siguiente, d, i - 1); }
    return sec;
}
```

Ejemplo 16.9. Supóngase una secuencia enlazada `sec` cuyos elementos están ordenados de menor a mayor. Se desea insertar un nuevo dato `d` en la secuencia, manteniéndola ordenada.

Para encontrar la ubicación del nuevo nodo, el siguiente código busca, usando una referencia `aux`, el primer elemento mayor o igual que `d`.

Acabada la búsqueda, la inserción debe distinguir si corresponde realizar una inserción en cabeza, o detrás de algún nodo. El primer caso se da cuando la búsqueda acaba con `aux == sec`: todos los elementos son mayores o iguales que `d`, lo que incluye el caso en que la secuencia esté vacía.

El segundo caso se da cuando en la secuencia hay al menos un elemento menor que `d`. Para facilitar la inserción, se usa una variable adicional `ant` que referencia en cada momento el nodo anterior a `aux`; al final, la inserción detrás de `ant` sitúa el nuevo nodo a continuación de todos los nodos con datos menores que `d`.

```
NodoInt aux = sec;
NodoInt ant = null; // el primer nodo no tiene anterior definido
while (aux != null && aux.dato < d) {
    ant = aux;
    aux = aux.siguiente;
}
if (aux == sec) { // ant == null
    sec = new NodoInt(d, sec);
} else { ant.siguiente = new NodoInt(d, aux); }
```

La versión recursiva se basa en el siguiente análisis de casos:

- Secuencia con $n = 0$ nodos, `sec == null`. Se inserta en cabeza de `sec`. El dato `d` es el primero que se inserta en `sec`.
- Secuencia con $n > 0$ nodos. Si `sec.dato >= d`, se inserta en cabeza de `sec`, sino el problema se reduce a insertar ordenadamente el dato `d` en la subsecuencia `sec.siguiente`.

Lo que da lugar al siguiente método:

```
public static NodoInt insertarOrd(NodoInt sec, int d) {
    if (sec == null) sec = new NodoInt(d);
    else {
        if (sec.dato >= d) { sec = new NodoInt(d, sec); }
        else { sec.siguiente = insertarOrd(sec.siguiente, d); }
    }
    return sec;
}
```

La eliminación de un nodo en una secuencia no vacía se resuelve igualmente sin necesidad de mover los otros datos. Como en la inserción, se distinguen dos casos:

- El nodo a eliminar es el primero de la secuencia (figura 16.13(a)):

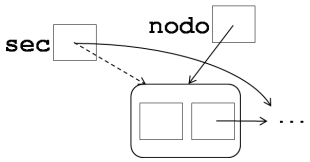
  ```
  sec = sec.siguiente;
  ```

  En el caso particular de que sólo hubiera un nodo, `sec` se haría `null`, es decir, la secuencia vacía.

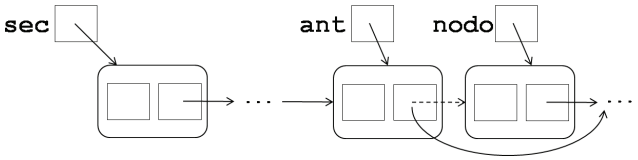
- El nodo a eliminar tiene un anterior:

  ```
  ant.siguiente = nodo.siguiente;
  ```

  en donde `nodo` es una referencia al nodo a eliminar y `ant` es una referencia al nodo anterior (figura 16.13(b)).



(a) Eliminación del primer nodo.



(b) Eliminación de cualquier otro nodo.

Figura 16.13: Eliminación de un nodo.

Ejemplo 16.10. Dada una secuencia enlazada `sec` se desea eliminar, si existe, la primera ocurrencia de un dato `d`. Si dicho dato no aparece, no se hace nada.

En primer lugar, mediante un esquema de búsqueda se localiza, si existe, el nodo cuyo dato es `d`. Si la búsqueda acaba con éxito, se elimina el nodo encontrado diferenciando si dicho nodo es el primero o tiene un anterior.

```
NodoInt aux = sec, ant = null;
while (aux != null && aux.dato != d) {
    ant = aux; aux = aux.siguiente;
}
if (aux != null) { // Éxito en la búsqueda
    if (ant == null) { // aux es el primer nodo
        sec = aux.siguiente;
    } else { ant.siguiente = aux.siguiente; }
}
```

El siguiente análisis de casos da forma recursiva al algoritmo:

- Secuencia con $n = 0$ nodos, `sec == null`. El dato `d` no está en `sec`, no se realiza ningún borrado.
- Secuencia con $n > 0$ nodos. Si se encuentra `d` en cabeza, se borra el primer nodo de `sec`; sino el problema se reduce a borrar la primera ocurrencia de `d` en la subsecuencia `sec.siguiente`.

Lo que conduce al siguiente método:

```
public static NodoInt borrar(NodoInt sec, int d) {
    if (sec != null) {
        if (sec.dato == d) { sec = sec.siguiente; }
        else { sec.siguiente = borrar(sec.siguiente, d); }
    }
    return sec;
}
```

Ejemplo 16.11. Se tiene una secuencia enlazada `sec` con valores enteros. El siguiente código elimina todos los valores menores que un cierto `umbral`.

```
NodoInt aux = sec, ant = null;
while (aux != null) {
    if (aux.dato < umbral) {
        if (aux == sec) { sec = sec.siguiente; }
        else { ant.siguiente = aux.siguiente; }
    }
    else { ant = aux; }
    aux = aux.siguiente;
}
```

El siguiente método resuelve el problema recursivamente. Este método se basa en un análisis por casos semejante al del ejemplo 16.10, excepto que en el caso general, además de eliminar si procede el primer nodo, siempre hay que completar recursivamente el borrado en la subsecuencia `sec.siguiente`.

```
public static NodoInt borrarMenores(NodoInt sec, int umbral) {
    if (sec != null) {
        NodoInt result = borrarMenores(sec.siguiente, umbral);
        if (sec.dato < umbral) { sec = result; }
        else { sec.siguiente = result; }
    }
    return sec;
}
```

Ejemplo 16.12. Para representar una secuencia enlazada de círculos, cada nodo tiene un dato de tipo `Circulo` (véase la figura 13.4 del capítulo 13), que forma parte del paquete `figuras`, y la clase `NodoCirculo` se define como se muestra en la figura 16.14.

```
import figuras.Circulo;
/**
 *  Clase NodoCirculo: representa un nodo de una secuencia enlazada
 *  que tiene un dato de tipo Circulo y un enlace al siguiente nodo.
 *  @author Libro IIP-PRG
 *  @version 2016
 */
class NodoCirculo {
    Circulo dato;
    NodoCirculo siguiente;

    /** Crea un nodo con un circulo c y al que sigue el nodo s.
     *  @param c Circulo, el dato del nuevo nodo.
     *  @param s NodoCirculo, con el que enlazar el nuevo nodo.
     */
    NodoCirculo(Circulo c, NodoCirculo s) {
        dato = c;
        siguiente = s;
    }

    /** Crea un nodo con un circulo c y sin siguiente.
     *  @param c Circulo, el dato del nuevo nodo.
     */
    NodoCirculo(Circulo c) { this(c, null); }
}
```

Figura 16.14: Clase `NodoCirculo`.

La clase `SecuenciaDeCirculosEnla` con igual funcionalidad que la clase `SecuenciaDeCirculos` (véase la figura 9.18 del capítulo 9) pero importando la clase `Circulo` del paquete `figuras` (visto en el capítulo 13), se puede definir como se muestra en la figura 16.15. Nótese que el método `insertar(Circulo)` tiene un coste lineal con el número de círculos de la secuencia que resulta poco apropiado y que se podría resolver con coste constante sin más que añadir un nuevo atributo `ultimo` que mantenga en todo momento la referencia al último nodo de la secuencia. Así, el método quedaría:

```
public void insertar(Circulo c) {
    NodoCirculo nuevo = new NodoCirculo(c);
    if (ultimo != null) { ultimo.siguiente = nuevo; }
    else { primero = nuevo; }
    ultimo = nuevo;
    talla++;
}
```

Se deja como ejercicio completar el resto de métodos para actualizar de forma apropiada este atributo `ultimo`.

```
import figuras.Circulo;
import java.awt.Color;
/**
 *  Clase SecuenciaDeCirculosEnla: representa una secuencia de círculos
 *  como una secuencia enlazada.
 *  @author Libro IIP-PRG
 *  @version 2016
 */
public class SecuenciaDeCirculosEnla {
    private int talla;           // número de círculos
    private NodoCirculo primero; // referencia al primer nodo

    /** Crea una secuencia vacía de círculos. */
    public SecuenciaDeCirculosEnla() {
        this.primero = null;
        this.talla = 0;
    }

    /** Añade un círculo al final de la secuencia.
     *  @param c Circulo, el círculo a añadir.
     */
    public void insertar(Circulo c) {
        NodoCirculo nuevo = new NodoCirculo(c);
        NodoCirculo aux = primero, ant = null;
        while (aux != null) {
            ant = aux;
            aux = aux.siguiente;
        }
        nuevo.siguiente = aux;
        if (ant == null) { primero = nuevo; }
        else { ant.siguiente = nuevo; }
        talla++;
    }

    /** Devuelve el círculo que ocupa la posición indicada.
     *  @param pos int, la posición dentro de la secuencia
     *         (el primer círculo ocupa la posición 0).
     *  @return Circulo, el círculo que ocupa la posición pos
     *                   o null si pos < 0 || pos >= talla.
     */
    public Circulo recuperar(int pos) {
        if (primero == null || pos < 0 || pos >= talla) { return null; }
        NodoCirculo aux = primero;
        for (int j = 0; j < pos; j++) { aux = aux.siguiente; }
        return aux.dato;
    }

    /** Devuelve el número de círculos de la secuencia.
     *  @return int, el número de círculos de la secuencia.
     */
    public int talla() { return this.talla; }
```

Figura 16.15: Clase `SecuenciaDeCirculosEnla`.

```
/** Elimina la primera aparición de un círculo de color
 *  col de la secuencia.
 *  @param col Color, el color del círculo a eliminar.
 *  @return boolean, true si se elimina con éxito o
 *          false si no hay ningún círculo de ese color.
 */
public boolean eliminar(Color col) {
    NodoCirculo aux = primero, ant = null;
    while (aux != null && !aux.dato.getColor().equals(col)) {
        ant = aux;
        aux = aux.siguiente;
    }
    if (aux != null) {
        if (ant == null) { primero = aux.siguiente; }
        else { ant.siguiente = aux.siguiente; }
        talla--;
        return true;
    }
    else { return false; }
}

/** Devuelve la posición de la primera aparición de un
 *  círculo de color col y radio r de la secuencia.
 *  @param col Color, el color del círculo a buscar.
 *  @param r double, el radio del círculo a buscar.
 *  @return int, la posición del círculo en la secuencia
 *             (el primer círculo ocupa la posición 0)
 *              o -1 si no está en la secuencia.
 */
public int indiceDe(Color col, double r) {
    int pos = -1, i = 0;
    for (NodoCirculo aux = primero; aux != null
        && pos == -1; aux = aux.siguiente) {
        if (aux.dato.getColor().equals(col)
            && aux.dato.getRadio() == r) { pos = i; }
        i++;
    }
    return pos;
}

/** Devuelve la suma de las áreas de todos los círculos
 *  de la secuencia.
 *  @return double, la suma de las áreas de todos los círculos.
 */
public double area() {
    double res = 0.0;
    for (NodoCirculo aux = primero; aux != null;
        aux = aux.siguiente) {
        res += aux.dato.area();
    }
    return res;
}
```

Figura 16.15: Clase `SecuenciaDeCirculosEnla` (cont.).

```
    /** Devuelve una descripción de la secuencia de círculos.
     *  @return String, la descripción de la secuencia de círculos.
     */
    public String toString() {
        String res = "Secuencia de " + talla + " círculos:\n";
        if (talla > 0) {
            for (NodoCirculo aux = primero; aux != null;
                 aux = aux.siguiente) {
                res += aux.dato.toString() + "\n";
            }
        } else { res = "Secuencia vacía\n"; }
        return res;
    }
}
```

Figura 16.15: Clase `SecuenciaDeCirculosEnla` (cont.).

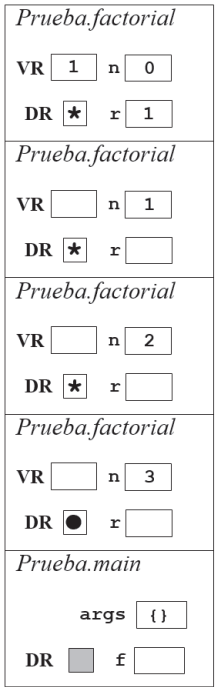
16.2 Tipos lineales

En esta sección se estudian los tipos de datos *Pila*, *Cola* y *Lista con punto de interés*, así como lo que se conoce como sus interfaces de operaciones, es decir, las funcionalidades respectivas de cada tipo. Para todos ellos se presentan dos posibles implementaciones en Java: una con arrays y otra con secuencias enlazadas. Dado que todas las clases definidas a continuación comparten la característica de linealidad y las versiones enlazadas de las implementaciones también comparten el uso de la clase auxiliar `NodoInt` definida como *friendly*, todas ellas se incluyen en un paquete denominado `lineales` para facilitar su localización y utilización y por ello, como puede verse, la primera línea de código de cada clase utilizada a continuación contiene la declaración: `package lineales;`

16.2.1 Pilas

Una pila (*stack* en inglés) es una secuencia en la que el acceso al primer elemento se realiza siguiendo un criterio LIFO (*Last In First Out*). Los elementos de una pila siempre se eliminan de ella en orden inverso al que fueron colocados, de modo que el último en entrar es el primero en salir y viceversa, el primero en entrar es el último en salir. Un ejemplo típico es la secuencia de registros de activación que coexisten en memoria, que se gestiona como una pila y de ahí el nombre que recibe la zona de memoria en la que se ubican estos registros (véase la figura 16.16).

El tipo de datos `Pila` presenta la siguiente interfaz de operaciones disponibles: crear una pila, apilar un nuevo elemento sobre la pila, desapilar el elemento que se encuentra en la cima de la pila, consultar (sin desapilar) el valor del dato que se encuentra en la cima de la pila, preguntar si una pila está vacía y, por último, obtener el número de elementos de la pila.




```
/** n >= 0 */
public static int factorial(int n) {
    int r;
    if (n == 0) { r = 1; }
    else { r = n * factorial(n - 1); } *
 ➡  return r;
}

public static void main(String[] args) {
    int f = factorial(3); ●
}
```

Figura 16.16: Pila de registros de activación.

En un contexto de programación orientada a objetos, podemos identificar la operación de crear una pila como un método constructor de la clase `Pila`. Las operaciones de conocer el dato que hay en la cima, saber si la pila está vacía o no y cuántos elementos contiene, se implementan por medio de métodos consultores que no alteran el estado de la pila. Finalmente, las operaciones de apilar y desapilar sí permiten modificar la estructura de la pila añadiendo o eliminando elementos, respectivamente, y se implementan por medio de métodos modificadores de la clase `Pila`. El interfaz genérico de una `Pila` se puede implementar en Java siguiendo el esquema de la tabla 16.1, donde `Tipo` representa el tipo de los datos de la `Pila`.

Es interesante constatar que todas estas operaciones se pueden implementar en Java de manera que cada método presente un coste constante, independientemente del número de elementos de la `Pila`.

Ejemplo 16.13. Se tiene una expresión `s` de tipo `String` en la que pueden aparecer subexpresiones encerradas entre parejas de `'('`, `')'`, `'{'`, `'}'`, siguiendo las reglas habituales de parentización. El siguiente método verifica si la expresión está bien o mal parentizada. Por ejemplo, el método devolvería `true` para `"{(a + b) / (c - a)} * (a - b)"` y `false` para `"{(a + b) / (c - a}) * (a - b)"`.

`public` **`Pila`**`()`	Crea una nueva `Pila` vacía.
`public void` **`apilar`**`(Tipo elemento)`	Apila el `elemento` sobre la `Pila`.
`public Tipo` **`desapilar`**`()`	Desapila el elemento de la cima de la `Pila` y lo devuelve. Lanza `NoSuchElementException` si la `Pila` está vacía.
`public Tipo` **`cima`**`()`	Devuelve (sin desapilarlo) el elemento de la cima de la `Pila`. Lanza `NoSuchElementException` si la `Pila` está vacía.
`public boolean` **`esVacia`**`()`	Devuelve `true` si la `Pila` está vacía y `false` en caso contrario.
`public int` **`talla`**`()`	Devuelve el número de elementos ($n \geq 0$) de la `Pila`.

Tabla 16.1: Interfaz de la clase `Pila`.

```
public static boolean expresionBienParentizada(String s) {
    Pila p = new Pila(); // los elementos de p son int
    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        if (c == '(' || c == '{') { p.apilar(i); }
        else if (c == ')' || c == '}') {
            if (!p.esVacia()) { // quedan () o {} por cerrar
                char parC = s.charAt(p.desapilar());
                if ((parC == '(' && c == '}')
                    || (parC == '{' && c == ')')) {
                    return false;
                }
            } else { return false; }
        }
    }
    if (p.esVacia()) { // se han cerrado todos los () o {}
        return true;
    } else { return false; }
}
```

El método se basa en que, leyendo la expresión de izquierda a derecha, sólo puede aparecer un paréntesis de cierre de un determinado tipo si el último paréntesis de apertura pendiente de ser cerrado es del mismo tipo.

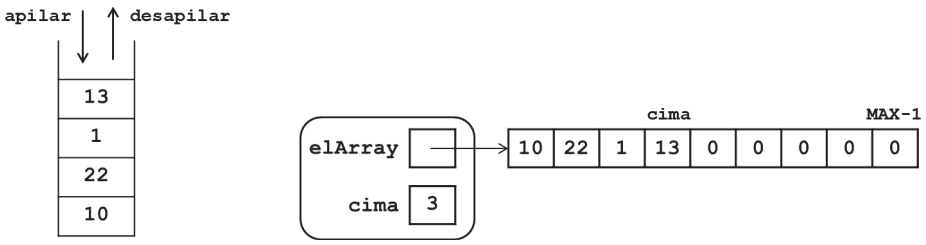
Para ello, se realiza un recorrido ascendente de la expresión y cada vez que se encuentra un paréntesis de apertura se apila su posición sobre una pila (creada inicialmente de un cierto tipo `Pila` cuyos elementos son de tipo `int`). De esta manera, en la cima de la pila se tiene cuál es el último paréntesis de apertura pendiente de ser cerrado.

En cambio, si se encuentra un paréntesis de cierre, para que la expresión esté bien parentizada el paréntesis de la cima de la pila debe ser el correspondiente paréntesis de apertura, en cuyo caso se extrae de la pila y se da por cerrado. Si no es así y la pila está vacía o el paréntesis encontrado no es el apropiado, entonces se concluye que la expresión está mal parentizada.

Cuando se termina el recorrido de toda la expresión, para que la expresión estuviera bien parentizada todos los paréntesis que se hubieran apilado se tendrán que haber cerrado con su correspondiente pareja, dejando finalmente la pila vacía.

Implementación mediante arrays

La clase `Pila` se puede definir mediante los siguientes campos o atributos: un array (`elArray`) para almacenar los datos, un índice al mismo (`cima`) que señale la cima de la pila (los datos se disponen consecutivamente entre las posiciones 0 y `cima`), y una constante que defina la dimensión inicial del array (`DIMENSION_INICIAL`). La pila de la figura 16.17(a) se representa gráficamente mediante esta estructura en la figura 16.17(b).



(a) Pila. (b) Pila con array.

Figura 16.17: Esquema gráfico de una pila mediante un array.

El código que se muestra en la figura 16.18 representa una `Pila`, usando un array, cuyos datos son de tipo `int`.

Inicialmente, el método constructor permite crear una pila vacía. Para ello, se crea el array de tamaño `DIMENSION_INICIAL` y se asigna un valor de `-1` al índice de la cima de la pila. Esto facilita la implementación del método `apilar(int)`, entre otros, ya que se puede programar el mismo algoritmo a la hora de apilar un nuevo elemento, independientemente de si anteriormente la pila estaba vacía o no, es decir, en ambos casos se almacena en la posición `cima + 1`. El método `apilar(int)` puede invocar al método privado `duplicaArray()` que permite redimensionar la longitud de `elArray`, incrementando el espacio de memoria asignado al mismo.

Las operaciones de `desapilar()` y `cima()` lanzan la excepción *unchecked* `NoSuchElementException` si la pila está vacía. En caso contrario, ambas de-

vuelven el dato que se encuentra en el índice `cima`. La diferencia estriba en que `desapilar()` decrementa en uno el valor del índice `cima` (quedando una posición a la izquierda).

Nótese que al desapilar, el dato que estaba en la cima de la pila sigue estando físicamente almacenado en el array, pero como el índice `cima` se ha decrementado, dicho valor queda almacenado fuera de la región de interés correspondiente a la pila y, por lo tanto, es como si efectivamente se desapilase. Además, dicho valor se sobrescribirá si la pila crece de nuevo hasta ese tamaño.

Por último, determinar el número de elementos de la pila n y, por extensión, deducir si está vacía o no, sólo depende del índice `cima`, siendo $n =$ `cima + 1`, dado que todos los datos se encuentran en el array entre las posiciones [0..`cima`].

```
package lineales;
import java.util.NoSuchElementException;
/**
 *  Clase PilaIntArray: Pila de int. Implementación con arrays.
 *  @author Libro IIP-PRG
 *  @version 2016
 */
public class PilaIntArray {
    private static final int MAX = 100;
    private int[] elArray;
    private int cima;

    /** Crea una pila vacía. */
    public PilaIntArray() {
        elArray = new int[MAX];
        cima = -1;
    }

    /** Apila x en la cima de la pila.
     *  @param x int, el valor a apilar.
     */
    public void apilar(int x) {
        if (cima + 1 == elArray.length) { duplicaArray(); }
        cima++;
        elArray[cima] = x;
    }

    /** Duplica el tamaño de elArray manteniendo los elementos. */
    private void duplicaArray() {
        int[] aux = new int[2 * elArray.length];
        for (int i = 0; i < elArray.length; i++) { aux[i] = elArray[i]; }
        elArray = aux;
    }
```

Figura 16.18: Clase `PilaIntArray`.

```
    /** Desapila y devuelve el elemento de la cima de la pila.
     *  @return int, la cima de la pila antes de desapilar.
     *  @throws NoSuchElementException si la pila está vacía.
     */
    public int desapilar() {
        if (cima < 0) { throw new NoSuchElementException("Pila vacía"); }
        return elArray[cima--];
    }

    /** Devuelve el elemento en la cima de la pila.
     *  @return int, la cima de la pila.
     *  @throws NoSuchElementException si la pila está vacía.
     */
    public int cima() {
        if (cima < 0) { throw new NoSuchElementException("Pila vacía"); }
        return elArray[cima];
    }

    /** Comprueba si la pila está vacía.
     *  @return boolean, true si la pila está vacía
     *                   y false en caso contrario.
     */
    public boolean esVacia() { return (cima == -1); }

    /** Devuelve la talla de la pila.
     *  @return int, la talla.
     */
    public int talla() { return cima + 1; }
}
```

Figura 16.18: Clase `PilaIntArray` (cont.).

Implementación mediante representación enlazada

La implementación de una pila mediante una secuencia enlazada de nodos establece como atributos: un objeto `NodoInt` que representa la `cima` de la pila y un entero `talla` que indica el número total de datos que contiene. La pila de la figura 16.17(a) se representa gráficamente usando esta estructura en la figura 16.19.

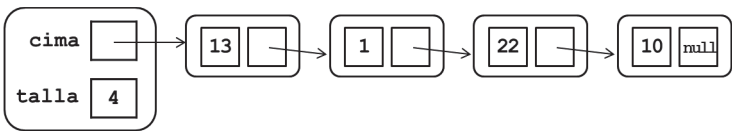


Figura 16.19: Esquema gráfico de una pila mediante representación enlazada.

El código de la `Pila` implementada en su versión enlazada se muestra en la figura 16.20.

El método constructor crea una pila vacía, es decir, una secuencia nula donde no hay ningún dato almacenado.

Para apilar un dato, se utiliza el constructor `NodoInt` más apropiado, en este caso, el que crea un nuevo nodo enlazándolo delante de la `cima` actual. El nuevo nodo introducido en la estructura es ahora la nueva `cima` de la pila.

Las operaciones de `desapilar()` y `cima()` lanzan la excepción *unchecked* `NoSuchElementException` si la pila está vacía. En caso contrario, ambas acceden al dato que se encuentra en el nodo `cima` y lo devuelven. En el caso de `desapilar()`, la referencia a la `cima` se actualiza al nodo `siguiente` de la `cima` actual. El resto de métodos (`esVacia()` y `talla()`) son sencillos de implementar.

```
package lineales;
import java.util.NoSuchElementException;
/**
 *  Clase PilaIntEnla: Pila de int. Implementación enlazada.
 *  @author Libro IIP-PRG
 *  @version 2016
 */
public class PilaIntEnla {
    private NodoInt cima;
    private int talla;

    /** Crea una pila vacía. */
    public PilaIntEnla() { cima = null;  talla = 0; }

    /** Apila x en la cima de la pila.
     *  @param int x, el valor a apilar.
     */
    public void apilar(int x) {
        cima = new NodoInt(x, cima);
        talla++;
    }

    /** Desapila y devuelve el elemento de la cima de la pila.
     *  @return int, la cima de la pila antes de desapilar.
     *  @throws NoSuchElementException si la pila está vacía.
     */
    public int desapilar() {
        if (cima == null) {
            throw new NoSuchElementException("Pila vacía");
        }
        int x = cima.dato;
        cima = cima.siguiente;
        talla--;
        return x;
    }
```

Figura 16.20: Clase `PilaIntEnla`.

```
    /** Devuelve el elemento en la cima de la pila.
     *  @return int, la cima de la pila.
     *  @throws NoSuchElementException si la pila está vacía.
     */
    public int cima() {
        if (cima == null) {
            throw new NoSuchElementException("Pila vacía");
        }
        return cima.dato;
    }

    /** Comprueba si la pila está vacía.
     *  @return boolean, true si la pila está vacía
     *                 y false en caso contrario.
     */
    public boolean esVacia() { return (cima == null); }

    /** Devuelve la talla de la pila.
     *  @return int, la talla.
     */
    public int talla() { return talla; }
}
```

Figura 16.20: Clase `PilaIntEnla` (cont.).

Comparación de implementaciones

La complejidad temporal de todas las operaciones en ambas implementaciones es constante e independiente del tamaño del problema: $T(n) \in \Theta(1)$.

En cuanto a la complejidad espacial, la implementación con arrays presenta el inconveniente de tener que estimar adecuadamente el tamaño máximo del array, y además, la reserva de un espacio que en muchos casos no se utilizará. Este consumo adicional de espacio no tendrá demasiada importancia si el tipo de las componentes del array es relativamente pequeño, como es el caso de una pila de enteros o de objetos (las componentes del array son referencias).

Por otro lado, la representación enlazada requiere un espacio de memoria adicional para almacenar los enlaces.

16.2.2 Colas

Una cola (*queue* en inglés) es una colección de datos del mismo tipo en la que el acceso se realiza siguiendo un criterio FIFO (*First In First Out*), es decir, el primer elemento que llega (que entra en la cola) es el primero en ser atendido (en ser eliminado de la cola).

En la vida real, las colas se utilizan muy a menudo como política de gestión de un modelo de negocio cliente-servidor. Por ejemplo, los usuarios de un comercio suelen hacer *cola* frente a las cajas a la hora de comprar un producto. Los ordenadores, a su vez, también gestionan muchos procesos mediante colas como, por ejemplo, la impresión de documentos (véase la figura 16.21).

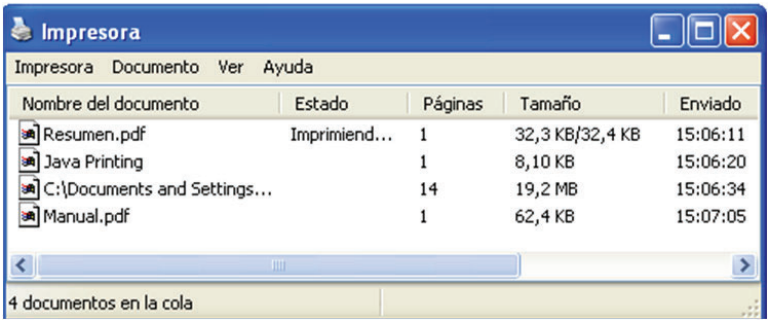


Figura 16.21: Cola de impresión.

El tipo de datos `Cola` presenta la siguiente interfaz de operaciones disponibles: crear una cola, añadir un nuevo elemento al final de la cola, eliminar el elemento que se encuentra a la cabeza de la cola, consultar (sin eliminar) el valor del dato que está el primero de la cola, preguntar si una cola está vacía y, por último, obtener el número de elementos de la cola. Siguiendo una aproximación similar a la de la clase `Pila`, este interfaz se implementa en Java por medio de:

- Un método constructor de objetos `Cola`.
- Sendos métodos modificadores para los procesos de encolar y desencolar.
- Varios métodos consultores que no alteran la estructura de una `Cola`.

El interfaz genérico de una `Cola` se implementa en Java siguiendo el esquema de la tabla 16.2, donde `Tipo` representa el tipo de los datos de la `Cola`.

Nótese que en castellano los verbos *encolar* y *desencolar* no tienen nada que ver con las estructuras de tipo `Cola` sino con pegar o despegar algún adhesivo; no obstante, resulta habitual su utilización en entornos de programación en el sentido de añadir y eliminar de una cola que, por simplicidad y equivalencia con la interfaz de las pilas, es el uso que se le da en este texto.

Nuevamente, se debe constatar que todas estas operaciones se pueden implementar en Java de manera que cada método presente un coste constante, independientemente del número de elementos de la cola.

`public` **Cola**`()`	Crea una nueva `Cola` vacía.
`public void` **encolar**`(Tipo elemento)`	Inserta el `elemento` al final de la `Cola`.
`public Tipo` **desencolar**`()`	Extrae el elemento de la cabeza de la `Cola` y lo devuelve. Lanza `NoSuchElementException` si la `Cola` está vacía.
`public Tipo` **primero**`()`	Devuelve (sin desencolarlo) el elemento de la cabeza de la `Cola`. Lanza `NoSuchElementException` si la `Cola` está vacía.
`public boolean` **esVacia**`()`	Devuelve `true` si la `Cola` está vacía y `false` en caso contrario.
`public int` **talla**`()`	Devuelve el número de elementos ($n \geq 0$) de la `Cola`.

Tabla 16.2: Interfaz de la clase `Cola`.

Ejemplo 16.14. El siguiente método devuelve `true` si el elemento `x` se encuentra en la cola de enteros `c` y `false` en caso contrario. A pesar de que la semántica del método representa un concepto de búsqueda, la resolución precisa realizar un recorrido, ya que el método debe preservar la estructura de la cola, restaurándola a su estado original tras las modificaciones introducidas por el propio método durante el proceso de búsqueda.

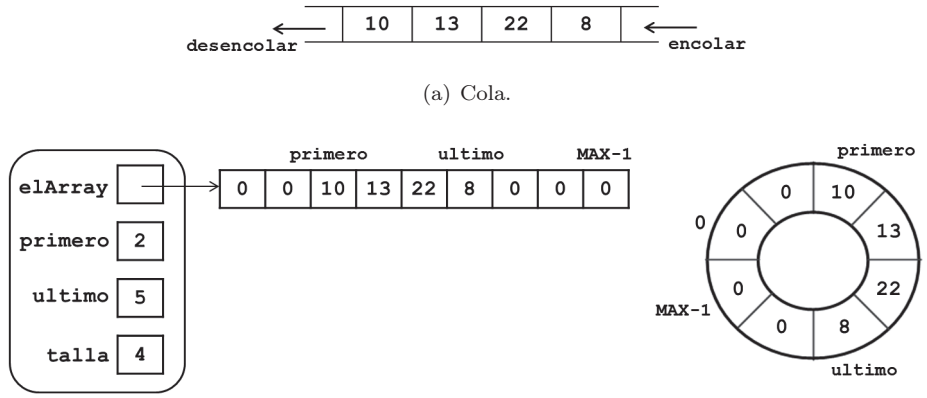
```
public static boolean buscar(Cola c, int x) {
    boolean exito = false;
    for (int i = 0; i < c.talla(); i++) {
        int dato = c.desencolar();
        if (dato == x) { exito = true; }
        c.encolar(dato);
    }
    return exito;
}
```

Implementación mediante arrays

La clase `Cola` se puede definir mediante los siguientes campos o atributos: un array *circular*[1] (`elArray`) para almacenar los datos, dos índices al mismo (`primero` y `ultimo`) que señalen el principio y el final de la cola (los datos se disponen consecutivamente entre ambos índices del array), un contador del número de elementos en cola (`talla`), y una constante que defina la dimensión inicial del array

[1] Para reutilizar todas las posiciones del array sin desplazar elementos, se considera el array como si fuera circular, sin principio ni fin, donde tras la posición $n-1$ va la posición 0.

(`DIMENSION_INICIAL`). En la figura 16.22(b) se representa gráficamente el estado de los atributos de la cola de la figura 16.22(a). En la 16.22(c) se muestra la estructura circular del atributo `elArray`.



(a) Cola.

(b) Cola con array.

(c) Array circular.

Figura 16.22: Esquema gráfico de una cola mediante arrays.

En la figura 16.23 se muestra el código de una `Cola`, usando un array, de datos de tipo `int`.

Inicialmente, el método constructor permite crear una cola vacía. Para ello, se crea el array de tamaño `DIMENSION_INICIAL` y se asigna un valor de `-1` al índice del último de la cola. Esto facilita la implementación de otros métodos ya que, por ejemplo, siempre se encola en la posición `(ultimo + 1) % elArray.length` independientemente del estado de la cola. Por este motivo, `primero` se ha de inicializar a `0`, para que al encolar el primer elemento, dicho dato sea a su vez el que esté a la cabeza de la cola (`primero`). El atributo `talla` representa el número de elementos de la cola n y, a partir del mismo, también se puede deducir si la cola está vacía o no. Se inicializa a `0` por razones obvias.

El método `encolar(int)` puede invocar al método privado `duplicaArray()` que permite redimensionar la longitud de `elArray`, incrementando el espacio de memoria asignado al mismo.

Las operaciones `desencolar()` y `primero()` lanzan la excepción *unchecked* `NoSuchElementException` si la cola está vacía. En caso contrario, ambas devuelven el dato que se encuentra en el índice `primero`. La diferencia estriba en que `desencolar()` incrementa en uno el valor del índice `primero` (quedando una posición a la derecha).

La implementación de los métodos `esVacia()` y `talla()` es obvia.

```
package lineales;
import java.util.NoSuchElementException;
/**
 *  Clase ColaIntArray: Cola de int.
 *  Implementación con arrays (array circular).
 *  @author Libro IIP-PRG
 *  @version 2016
 */
public class ColaIntArray {
    private static final int MAX = 100;
    private int[] elArray;
    private int primero, ultimo, talla;

    /** Crea una cola vacía. */
    public ColaIntArray() {
        elArray = new int[MAX];
        talla = 0;
        primero = 0;
        ultimo = -1;
    }

    /** Encola un nuevo elemento en la cola.
     *  @param int x, el elemento a encolar.
     */
    public void encolar(int x) {
        if (talla == elArray.length) { duplicaArray(); }
        ultimo = (ultimo + 1) % elArray.length;
        elArray[ultimo] = x;
        talla++;
    }

    /** Duplica el tamaño del elArray manteniendo los elementos. */
    private void duplicaArray() {
        int[] aux = new int[2 * elArray.length];
        int pos = primero;
        for (int i = 0; i < elArray.length; i++) {
            aux[i] = elArray[pos];
            pos = (pos + 1) % elArray.length;
        }
        primero = 0;
        ultimo = elArray.length - 1;
        elArray = aux;
    }
```

Figura 16.23: Clase `ColaIntArray`.

```
    /** Desencola y devuelve el primer elemento de la cola.
     *  @return int, la cabeza de la cola antes de desencolar.
     *  @throws NoSuchElementException si la cola está vacía.
     */
    public int desencolar() {
        if (talla == 0) {
            throw new NoSuchElementException("Cola vacía");
        }
        int x = elArray[primero];
        primero = (primero + 1) % elArray.length;
        talla--;
        return x;
    }

    /** Devuelve el primer elemento de la cola.
     *  @return int, la cabeza de la cola.
     *  @throws NoSuchElementException si la cola está vacía.
     */
    public int primero() {
        if (talla == 0) {
            throw new NoSuchElementException("Cola vacía");
        }
        return elArray[primero];
    }

    /** Comprueba si la cola está vacía.
     *  @return boolean, true si la cola está vacía
     *                 y false en caso contrario.
     */
    public boolean esVacia() { return (talla == 0); }

    /** Devuelve la talla de la cola.
     *  @return int, la talla.
     */
    public int talla() { return talla; }
}
```

Figura 16.23: Clase `ColaIntArray` (cont.).

Implementación mediante representación enlazada

La implementación de una cola mediante una secuencia enlazada de nodos establece como atributos dos objetos `NodoInt` representando el inicio (`primero`) y el final (`ultimo`) de la cola, respectivamente, y un entero `talla` que indica el número total de datos que contiene. El acceso directo al último nodo de la secuencia permite poder `encolar` un nuevo nodo detrás del mismo sin tener que recorrer toda la secuencia desde el primero de sus nodos, es decir, implementar dicha operación con un coste constante, independiente de la talla. La cola de la figura 16.22(a) se representa gráficamente usando esta estructura en la figura 16.24.

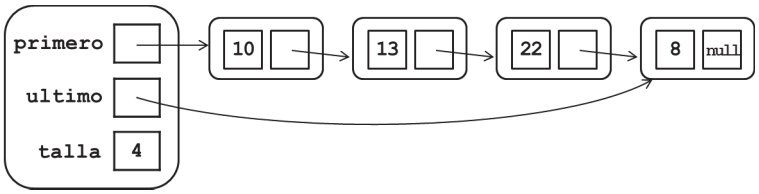


Figura 16.24: Esquema gráfico de una cola mediante representación enlazada.

En la figura 16.25 se muestra el código de una `Cola`, con representación enlazada, de datos de tipo `int`.

El método constructor crea una cola vacía, es decir, una secuencia nula donde no hay ningún dato almacenado. Para encolar un nuevo dato, hay que diferenciar si la cola está vacía o no. Si no está vacía, el nuevo nodo se ha de enlazar a continuación del último nodo. En caso contrario, la cola está vacía y, por tanto, el nuevo nodo es el `primero`. En todo caso, el nuevo nodo será ahora el que ocupe el último lugar en la cola.

Las operaciones `desencolar()` y `primero()` lanzan la excepción *unchecked* `NoSuchElementException` si la cola está vacía. En caso contrario, ambas acceden al dato que se encuentra en el nodo `primero`, devolviéndolo. Al desencolar, la referencia a `primero` se actualiza al nodo `siguiente` a éste. Si sólo hubiera 1 elemento en la cola (los nodos `primero` y `ultimo` coinciden), `desencolar()` afecta asimismo al atributo `ultimo`, dejándolo también como `null`.

Los métodos `esVacia()` y `talla()` no precisan discusión.

Además de las operaciones de la interfaz, se han incluido los métodos `toString()` y `equals(Object)` (que sobrescriben a los de `Object`), con coste lineal con la talla de la lista. Se deja como ejercicio la implementación de estos métodos en la implementación mediante arrays de la `Cola`.

Comparación de implementaciones

La complejidad temporal de todas las operaciones en ambas implementaciones es constante e independiente de la talla del problema: $T(n) \in \Theta(1)$.

En cuanto a la complejidad espacial, al igual que sucedía con las pilas, la implementación con arrays presenta el inconveniente de estimar el tamaño del array y, además, la reserva de un espacio que en muchos casos no se utilizará. Pero del mismo modo, este consumo adicional de espacio no tendrá demasiada importancia si el tipo de las componentes del array es relativamente pequeño, como es el caso de una cola de números enteros o de objetos Java (referencias). En este caso también, la representación enlazada requiere un espacio de memoria adicional para almacenar los enlaces.

```
package lineales;
import java.util.NoSuchElementException;
/**
 *  Clase ColaIntEnla: Cola de int. Implementación enlazada.
 *  @author Libro IIP-PRG
 *  @version 2016
 */
public class ColaIntEnla {
    private NodoInt primero, ultimo;
    private int talla;

    /** Crea una cola vacía. */
    public ColaIntEnla() {
        primero  = null;
        ultimo = null;
        talla = 0;
    }

    /** Encola un nuevo elemento en la cola.
     *  @param int x, el elemento a encolar.
     */
    public void encolar(int x) {
        NodoInt nuevo = new NodoInt(x);
        if (ultimo != null) { ultimo.siguiente = nuevo; }
        else { primero = nuevo; }
        ultimo = nuevo;
        talla++;
    }

    /** Desencola y devuelve el primer elemento de la cola.
     *  @return int, la cabeza de la cola antes de desencolar.
     *  @throws NoSuchElementException si la cola está vacía.
     */
    public int desencolar() {
        if (talla == 0) {
            throw new NoSuchElementException("Cola vacía");
        }
        int x = primero.dato;
        primero = primero.siguiente;
        if (primero == null) { ultimo = null; }
        talla--;
        return x;
    }

    /** Devuelve el primer elemento de la cola.
     *  @return int, la cabeza de la cola.
     *  @throws NoSuchElementException si la cola está vacía.
     */
    public int primero() {
        if (talla == 0) {
            throw new NoSuchElementException("Cola vacía");
        }
        return primero.dato;
    }
```

Figura 16.25: Clase `ColaIntEnla`.

```
    /** Comprueba si la cola está vacía.
     *  @return boolean, true si la cola está vacía
     *                  y false en caso contrario.
     */
    public boolean esVacia() { return (talla == 0); }

    /** Devuelve la talla de la cola.
     *  @return int, la talla.
     */
    public int talla() { return talla; }

    /** Devuelve un String con los datos de la cola.
     *  @return String, los datos de la cola.
     */
    public String toString() {
        String s = "";
        NodoInt aux = primero;
        while (aux != null) {
            s += String.format("%4d", aux.dato);
            aux = aux.siguiente;
        }
        return s;
    }

    /** Comprueba si la cola es igual o no a una cola dada.
     *  @param o Object, la cola a comparar.
     *  @return boolean, true si son iguales
     *                  y false en caso contrario.
     */
    public boolean equals(Object o) {
        boolean igual = false;
        if (o instanceof ColaIntEnla) {
            ColaIntEnla c = (ColaIntEnla) o;
            if (talla == c.talla) {
                NodoInt aux = primero, auxC = c.primero;
                while (aux != null && aux.dato == auxC.dato) {
                    aux = aux.siguiente;
                    auxC = auxC.siguiente;
                }
                if (aux == null) { igual = true; }
            }
        }
        return igual;
    }
}
```

Figura 16.25: Clase `ColaIntEnla` (cont.).

16.2.3 Listas con punto de interés

Una lista es una secuencia en la que el acceso se puede realizar en cualquier punto. En una lista se puede buscar un elemento, insertar, o eliminar un elemento, en una posición dada.

Además, en una lista suele existir el concepto de posición o elemento activo, conocido como el *punto de interés* (o *cursor*) de la lista. Si la lista tiene n datos, $n \geq 0$, numerados de 0 a $n-1$, el cursor puede estar:

- en una posición $0 \leq i \leq n-1$: *sobre el elemento i-ésimo*,
- en la posición $i = n$: *a la derecha del todo*, detrás del último elemento.

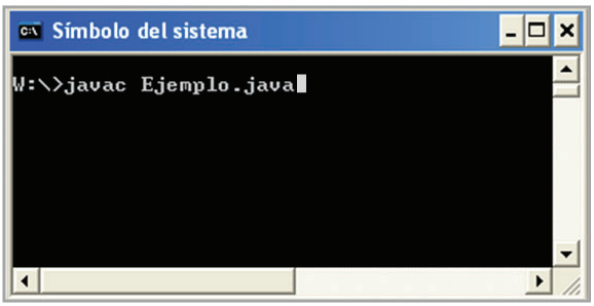
Dicho cursor se puede mover, posición a posición, a lo largo de la lista.

El ejemplo paradigmático de lista con punto de interés o cursor es la línea de comandos del sistema (figura 16.25(a)), que se edita como una lista de caracteres en la que se puede borrar el carácter remarcado por el cursor, o insertar un carácter por delante del cursor, etc. Por ejemplo:

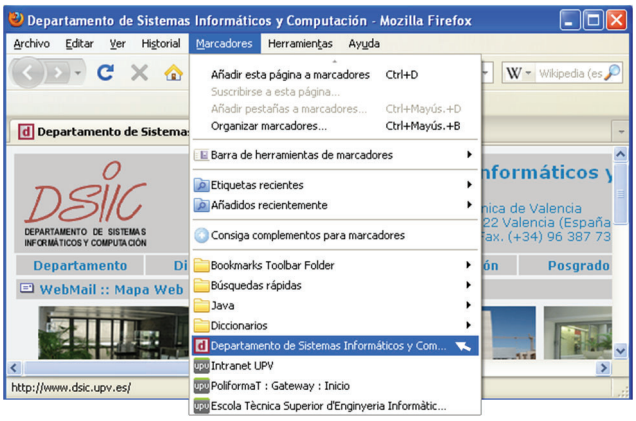
```
W:\ _
W:\ E_
W:\ Ej_
W:\ Eje_
W:\ Ejen_
W:\ Ejen
W:\ Eje_
W:\ Ejem_
```

Otros ejemplos de listas que usan un cursor para acceder a sus elementos son las listas de opciones de un menú desplegable, como el de la figura 16.25(b).

En el modelo que se va a presentar, el movimiento del cursor se limita exclusivamente a avanzarlo una posición a la derecha o hacerlo retroceder al inicio. El conjunto de operaciones del tipo disponibles es el siguiente: crear una lista, situar el punto de interés al principio de la lista, hacer avanzar una posición el punto de interés, consultar si el punto de interés está al final de la lista (detrás del último elemento), insertar un nuevo elemento antes del punto de interés, eliminar el elemento que se encuentra en el punto de interés, consultar (sin eliminarlo) el valor del dato que está en el punto de interés, obtener el número de elementos de la lista y, por último, preguntar si una lista está vacía.



(a) Línea de comandos del sistema.



(b) Lista de opciones de un menú desplegable.

Figura 16.25: Listas con cursor o punto de interés.

Siguiendo una aproximación similar a la de las clases `Pila` y `Cola`, este interfaz se implementa en Java por medio de:

- Un método constructor de objetos `Lista`.
- Dos métodos modificadores para desplazar el punto de interés.
- Otros dos métodos modificadores para los procesos de insertar y eliminar.
- Varios métodos consultores que no alteran la estructura de una `Lista`.

El interfaz genérico de una `Lista` se implementa en Java siguiendo el esquema de la tabla 16.3, donde `Tipo` representa el tipo de los datos de la `Lista`.

`public` **Lista**`()`	Crea una nueva `Lista` vacía.
`public void` **inicio**`()`	Sitúa el cursor al principio de la `Lista`.
`public void` **siguiente**`()`	Desplaza el cursor una posición a la derecha de su posición actual. Lanza `NoSuchElementException` si el cursor está al final de la `Lista`.
`public void` **insertar**`(Tipo elemento)`	Inserta el `elemento` antes del punto de interés de la `Lista`.
`public Tipo` **eliminar**`()`	Elimina el elemento del punto de interés de la `Lista`, devolviéndolo. Lanza `NoSuchElementException` si el cursor está al final de la `Lista`.
`public Tipo` **recuperar**`()`	Devuelve (sin eliminarlo) el elemento que está en el cursor. Lanza `NoSuchElementException` si el cursor está al final de la `Lista`.
`public boolean` **esFin**`()`	Devuelve `true` si el cursor está al final de la `Lista` y `false` en caso contrario.
`public boolean` **esVacia**`()`	Devuelve `true` si la `Lista` está vacía y `false` en caso contrario.
`public int` **talla**`()`	Devuelve el número de elementos ($n \geq 0$) de la `Lista`.

Tabla 16.3: Interfaz de la clase `Lista`.

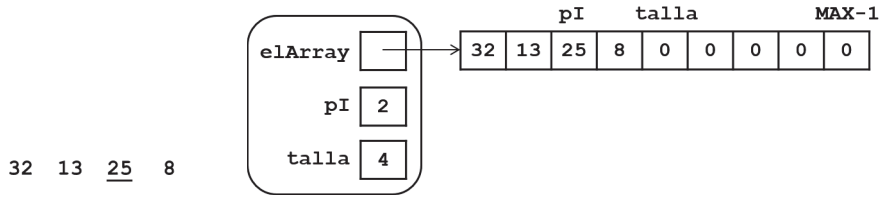
Ejemplo 16.15. Este es un ejemplo de uso de una lista con cursor o punto de interés de enteros a través de un esquema genérico de búsqueda de un elemento sobre la misma. El siguiente método devuelve `true` si el elemento `x` se encuentra en la lista `l`, situando el punto de interés en la primera ocurrencia de `x` en `l`. En caso contrario, el punto de interés se queda al final de `l` y devuelve `false`.

```
public static boolean buscar(Lista l, int x) {
    l.inicio();
    while (!l.esFin() && l.recuperar() != x) { l.siguiente(); }
    if (!l.esFin()) { return true; }
    else { return false; }
}
```

Implementación mediante arrays

La clase `Lista` se puede definir mediante los siguientes campos o atributos: un array (`elArray`) para almacenar los datos, un índice al mismo (`pI`) para representar el punto de interés, un contador (`talla`) del número de elementos en la lista (los

datos se disponen consecutivamente entre las posiciones 0 y `talla - 1`), y una constante que defina la dimensión inicial del array (`DIMENSION_INICIAL`). La lista de la figura 16.26(a) (donde el subrayado indica la posición del cursor) se representa gráficamente mediante esta estructura en la figura 16.26(b).



(a) Lista con punto de interés.

(b) Lista con punto de interés con array.

Figura 16.26: Esquema gráfico de una lista con punto de interés mediante arrays.

En la figura 16.27 se muestra el código de una Lista con punto de interés, con un array, de datos de tipo `int`.

Inicialmente, el método constructor permite crear una lista vacía. Para ello, se crea el array de tamaño `DIMENSION_INICIAL` y se asigna un valor de `0` al atributo `talla`. El índice asociado al punto de interés se inicializa al final de la lista (vacía, en este caso), cuyo valor es `talla`, ya que la lista se almacena en `[0..talla - 1]`.

Situar el punto de interés al principio de la lista no tiene mayor complejidad que ponerlo a cero.

Al insertar un nuevo dato en el punto de interés, hay que desplazar a la derecha todos los datos desde dicho punto de interés hasta el final de la lista. Esto implica que el coste en el peor caso sea lineal respecto a la talla de la lista. El punto de interés debe mantenerse en el mismo elemento. El método `insertar(int)` puede invocar, en caso necesario, al método privado `duplicaArray()` que permite redimensionar la longitud de `elArray`, incrementando el espacio de memoria asignado al mismo.

Las operaciones `eliminar()`, `recuperar()` y `siguiente()` lanzan la excepción *unchecked* `NoSuchElementException` si el punto de interés está al final, después del último elemento de la lista. En el caso de `eliminar()`, hay que desplazar a la izquierda todos los datos desde dicho punto de interés hasta el final de la lista. Esto implica nuevamente que el coste en el peor caso sea lineal respecto a la talla de la lista.

Saber si la lista está vacía o no, cuántos elementos $n \geq 0$ contiene la lista, o si el cursor está o no al final de la misma, son operaciones de consulta que dependen directamente del atributo `talla`.

```
package lineales;
import java.util.NoSuchElementException;
/**
 *  Clase ListaPIIntArray: Lista con punto de interés (o cursor) de int.
 *  Implementación (ineficiente) con array.
 *  @author Libro IIP-PRG
 *  @version 2016
 */
public class ListaPIIntArray {
    private static final int MAX = 100;
    private int[] elArray;
    private int pI, talla;

    /** Crea una ListaPIIntArray vacía. */
    public ListaPIIntArray() {
        elArray = new int[MAX];
        talla = 0;
        pI = 0;
    }

    /** Sitúa el cursor al inicio de la lista. */
    public void inicio() { pI = 0; }

    /** Desplaza el cursor una posición a la derecha.
     *  @throws NoSuchElementException si el cursor está al final.
     */
    public void siguiente() {
        if (pI == talla) {
            throw new NoSuchElementException("Cursor al final");
        }
        pI++;
    }

    /** Inserta x en la lista, delante del cursor.
     *  @param int x, el elemento a insertar.
     */
    public void insertar(int x) {
        if (talla == elArray.length) { duplicaArray(); }
        for (int k = talla - 1; k >= pI; k--) {
            elArray[k + 1] = elArray[k];
        }
        elArray[pI] = x;
        pI++;
        talla++;
    }

    /** Duplica el tamaño del elArray manteniendo los elementos. */
    private void duplicaArray() {
        int[] aux = new int[2 * elArray.length];
        for (int i = 0; i < elArray.length; i++) { aux[i] = elArray[i]; }
        elArray = aux;
    }
```

Figura 16.27: Clase `ListaPIIntArray`.

```
    /** Elimina y devuelve el elemento de la lista sobre el que está
     *  el cursor.
     *  @return int, el elemento situado en el cursor antes de eliminar.
     *  @throws NoSuchElementException si el cursor está al final.
     */
    public int eliminar() {
        if (pI == talla) {
            throw new NoSuchElementException("Cursor al final");
        }
        int x = elArray[pI];
        for (int k = pI + 1; k < talla; k++) {
            elArray[k - 1] = elArray[k];
        }
        talla--;
        return x;
    }

    /** Devuelve el elemento de la lista sobre el que está el cursor.
     *  @return int, el elemento situado en el cursor.
     *  @throws NoSuchElementException si el cursor está al final.
     */
    public int recuperar() {
        if (pI == talla) {
            throw new NoSuchElementException("Cursor al final");
        }
        return elArray[pI];
    }

    /** Comprueba si el cursor está al final de la lista.
     *  @return boolean, true si el cursor está al final
     *                   y false en caso contrario.
     */
    public boolean esFin() { return (pI == talla); }

    /** Comprueba si la lista está vacía.
     *  @return boolean, true si la cola está vacía
     *                   y false en caso contrario.
     */
    public boolean esVacia()   { return (talla == 0); }

    /** Devuelve la talla de la lista.
     *  @return int, la talla.
     */
    public int talla() { return talla; }
}
```

Figura 16.27: Clase `ListaPIIntArray` (cont.).

Implementación mediante representación enlazada

La implementación de una lista mediante una secuencia enlazada de nodos establece como atributos dos objetos `NodoInt` representando el primer nodo (`primero`) y el del punto de interés (`pI`) de la lista, respectivamente, y un entero `talla` que

indica el número total de datos almacenados en la misma. El esquema de eliminación de la sección 16.1.3 muestra que se requiere además un atributo (`antPI`) que apunte al nodo anterior al del punto de interés. Nótese que disponer del atributo `antPI` permitiría prescindir del atributo `pI`, dado que éste último está accessible a través de la expresión `antPI.siguiente` (véase el ejemplo 17). Sin embargo, para simplificar la discusión de las operaciones, se ha decidido mantener ambos atributos en esta representación. La representación enlazada propuesta de la lista con punto de interés de la figura 16.26(a) se representa gráficamente en la figura 16.28.

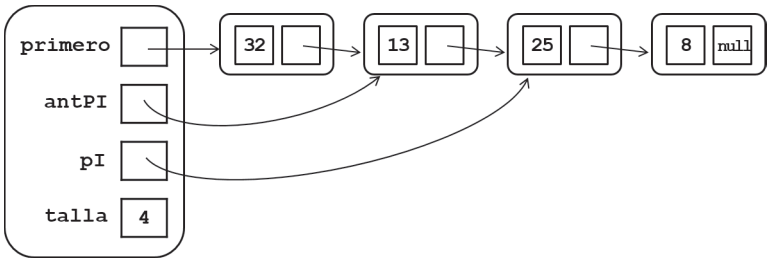


Figura 16.28: Esquema gráfico de una lista con punto de interés mediante representación enlazada.

En la figura 16.29 se muestra el código de una Lista con punto de interés con representación enlazada de datos de tipo `int`.

El método constructor crea una lista vacía, es decir, una secuencia nula donde no hay ningún dato almacenado.

Llevar el punto de interés al principio de la lista es una simple asignación entre la referencia del nodo cursor y la del primer nodo de la lista.

Al insertar un nuevo dato, hay que distinguir si el cursor está al inicio o no. Si el cursor está al inicio, el nuevo nodo será el primero de la lista. En cambio, en cualquier otra situación, éste se situará entre el nodo cursor y su predecesor. En ambos casos, el nuevo nodo será el nuevo nodo predecesor del nodo cursor y la talla se incrementará.

Las operaciones `eliminar()`, `recuperar()` y `siguiente()` lanzan la excepción *unchecked* `NoSuchElementException` si el punto de interés está al final, después del último elemento de la lista. En caso de `eliminar()` el primer dato de la lista, `primero` se actualiza al nodo `siguiente` a éste. En cualquier otro caso, los nodos anterior y posterior al nodo cursor quedan enlazados.

Las consultas sobre si la lista está vacía o no, o sobre cuántos elementos hay almacenados en total, son operaciones que dependen del atributo `talla`, mientras

que consultar si el punto de interés está o no al final de la lista se implementa como una comparación entre la referencia `pI` y el valor `null`.

```
package lineales;
import java.util.NoSuchElementException;
/**
 *  Clase ListaPIIntArray: Lista con punto de interés (o cursor) de int.
 *  Implementación con secuencia enlazada.
 *  @author Libro IIP-PRG
 *  @version 2016
 */
public class ListaPIIntEnla {
    private NodoInt primero, pI, antPI;
    private int talla;

    /** Crea una ListaPIIntEnla vacía. */
    public ListaPIIntEnla() {
        primero = null;
        pI = null;
        antPI = null;
        talla = 0;
    }

    /** Sitúa el cursor al inicio de la lista.  */
    public void inicio() { pI = primero; antPI = null; }

    /** Desplaza el cursor una posición a la derecha.
     *  @throws NoSuchElementException si el cursor está al final.
     */
    public void siguiente() {
        if (pI == null) {
            throw new NoSuchElementException("Cursor al final");
        }
        antPI = pI;
        pI = pI.siguiente;
    }

    /** Inserta x en la lista, delante del cursor.
     *  @param int x, el elemento a insertar.
     */
    public void insertar(int x) {
        if (pI == primero) {
            primero = new NodoInt(x, pI);
            antPI = primero;
        } else {
            antPI.siguiente = new NodoInt(x, pI);
            antPI = antPI.siguiente;
        }
        talla++;
    }
```

Figura 16.29: Clase `ListaPIIntEnla`.

```
    /** Elimina y devuelve el elemento de la lista sobre el que está
     *  el cursor.
     *  @return int, el elemento situado en el cursor antes de eliminar.
     *  @throws NoSuchElementException si el cursor está al final.
     */
    public int eliminar() {
        if (pI == null) {
            throw new NoSuchElementException("Cursor al final");
        }
        int x = pI.dato;
        if (pI == primero) { primero = primero.siguiente; }
        else { antPI.siguiente = pI.siguiente; }
        pI = pI.siguiente;
        talla--;
        return x;
    }

    /** Devuelve el elemento de la lista sobre el que está el cursor.
     *  @return int, el elemento situado en el cursor.
     *  @throws NoSuchElementException si el cursor está al final.
     */
    public int recuperar() {
        if (pI == null) {
            throw new NoSuchElementException("Cursor al final");
        }
        return pI.dato;
    }

    /** Comprueba si el cursor está al final de la lista.
     *  @return boolean, true si el cursor está al final
     *                 y false en caso contrario.
     */
    public boolean esFin() { return (pI == null); }

    /** Comprueba si la lista está vacía.
     *  @return boolean, true si la cola está vacía
     *                 y false en caso contrario.
     */
    public boolean esVacia()   { return (talla == 0); }

    /** Devuelve la talla de la lista.
     *  @return int, la talla.
     */
    public int talla() { return talla; }
}
```

Figura 16.29: Clase `ListaPIIntEnla` (cont.)

Comparación de implementaciones

La complejidad temporal de las operaciones `insertar` y `eliminar` difiere según la implementación: el coste es, en el caso peor, lineal con la talla de la lista en la implementación mediante arrays, mientras que dicho coste es constante en la implementación de nodos enlazados. El resto de operaciones presentan un coste temporal constante en ambas implementaciones.

La representación naíf con arrays presentada en esta sección tiene una alternativa más eficiente, mostrada en la figura 16.30. Esta representación usa dos arrays que funcionan a modo de pilas, de manera que si en la lista $d_0 d_1 \ldots d_{n-1}$ el `pI` se encuentra sobre un cierto d_i:

- `elArrayIzq[0..antPI]` contiene los elementos de $d_0 d_1 \ldots d_{i-1}$,
- `elArrayDer[0..pI]` contiene los elementos de $d_{n-1} d_{n-2} \ldots d_i$.

Con ello, la inserción delante del `pI` se resuelve con coste $\Theta(1)$ situando el nuevo dato en `elArrayDer[antPI + 1]` y actualizando dicho índice `antPI`, mientras que la eliminación del elemento en el `pI` se resuelve, también con coste $\Theta(1)$, simplemente decrementando el índice `pI`.

El movimiento del `pI` al siguiente elemento de la lista se resuelve con el trasvase de `elArrayDer[pI]` a `elArrayIzq[antPI + 1]` y la correspondiente actualización de ambos índices; además, se permitiría implementar el desplazamiento al anterior elemento con un trasvase análogo, pero en sentido contrario.

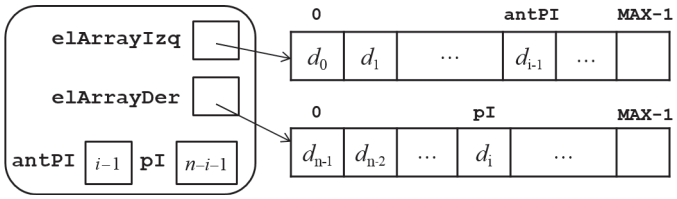


Figura 16.30: Representación mediante dos arrays de una lista con punto de interés.

Cabe señalar que la operación de movimiento del `pI` de un elemento al anterior también se podría resolver de modo directo, con coste $\Theta(1)$, con una implementación de *secuencias doblemente enlazadas*, en las que los nodos contuviesen referencias a los nodos anterior y siguiente (véase figura 16.31).

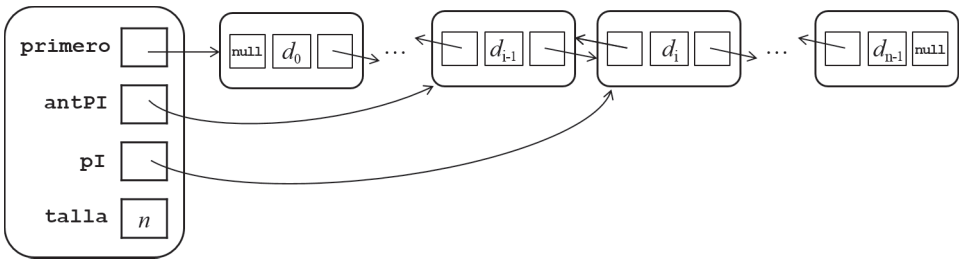


Figura 16.31: Representación mediante una secuencia doblemente enlazada de una lista con punto interés.

En cualquier caso, la discusión sobre el coste espacial es la misma que para las pilas y colas. Se deja como ejercicio diseñar una clase que implemente una lista con punto de interés de enteros mediante una secuencia doblemente enlazada.

Ejemplo 16.16. El siguiente método ampliaría la funcionalidad de la interfaz `Lista` permitiendo situar el cursor al final de la lista, esto es, después de su último elemento:

```
public void fin() {
    pI = primero; antPI = null;
    while (pI != null) {
        antPI = pI;
        pI = pI.siguiente;
    }
}
```

El coste de este método es lineal con el número de datos de la lista. Se puede implementar con coste constante si en la clase `ListaPIIntEnla` se añade un nuevo atributo `ultimo` de tipo `NodoInt` que mantiene la referencia al último nodo de la lista. Con este atributo, el código para el método anterior será:

```
public void fin() {
    antPI = ultimo;
    pI = null;
}
```

Se deja como ejercicio la modificación de los métodos constructor, `insertar(int)` y `eliminar()` para que actualicen de forma adecuada dicho atributo.

Ejemplo 16.17. La implementación enlazada básica de listas con punto de interés contempla dos atributos (`pI` y `antPI`) para referenciar al nodo en el punto de interés y a su predecesor, respectivamente. Ya al comienzo de esta sección, se dice que la referencia `pI` es prescindible dado que dicho nodo es accesible como el `siguiente` del nodo `antPI`. Esto es así excepto en el caso de que el punto de interés se sitúe sobre el primer nodo. Aunque se puede tratar esta situación como un caso particular y desarrollar código específico para su tratamiento, una forma más sencilla de tratarla es utilizar un *nodo cabecera o ficticio* que no alberga ningún elemento de la lista y que siempre referencia al primer nodo.

La clase `ListaPIIntEnlaF` con la misma funcionalidad que `ListaPIIntEnla`, en la que el coste de todos sus métodos (incluido el método `fin`) también es constante, tendrá como atributos tres objetos `NodoInt` `primero`, `antPI` y `ultimo` y un entero `talla` que indica el número total de datos almacenados en la misma. El atributo `primero` referencia o apunta al primer nodo de la lista que es ficticio, no al que contiene el primer dato de la lista (`primero.siguiente`); `antPI` apunta al nodo

anterior al punto de interés de la lista, por lo que `antPI.siguiente` representa el punto de interés de la lista y `ultimo` apunta al último nodo de la lista (véase la figura 16.32 en la que aparece sombreado el nodo del punto de interés).

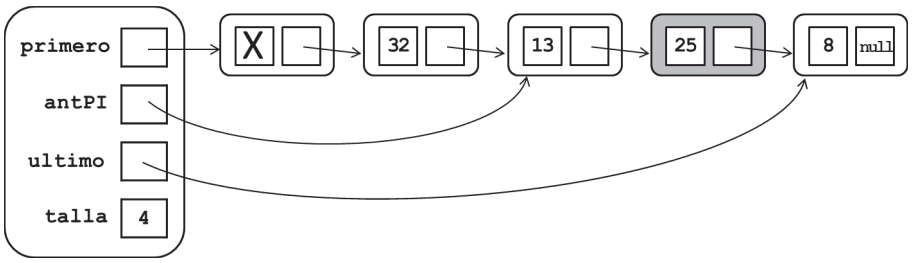


Figura 16.32: Esquema gráfico de una lista con punto de interés con `primero, antPI, ultimo` y nodo ficticio.

El método constructor ahora debe ejecutar la instrucción:

```
primero = ultimo = antPI = new NodoInt(Integer.MIN_VALUE);
```

Así, una lista con punto de interés se construye vacía, sin datos, creando una lista con un nodo cabecera o ficticio[2] al que apuntan los enlaces `primero`, `antPI` y `ultimo`, como se ve en la figura 16.33.

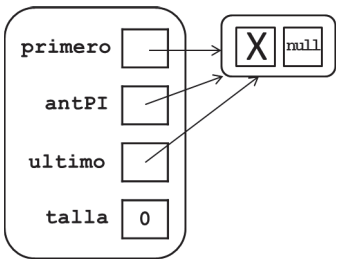
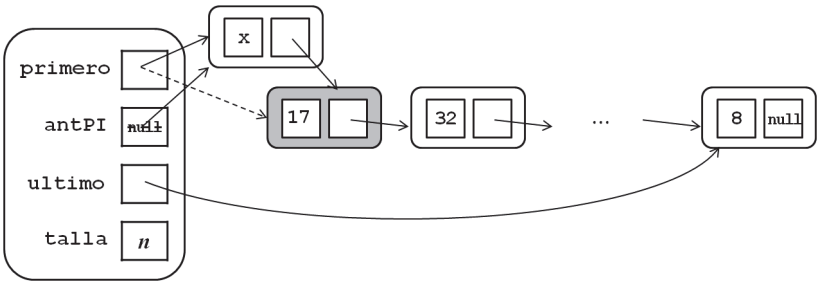


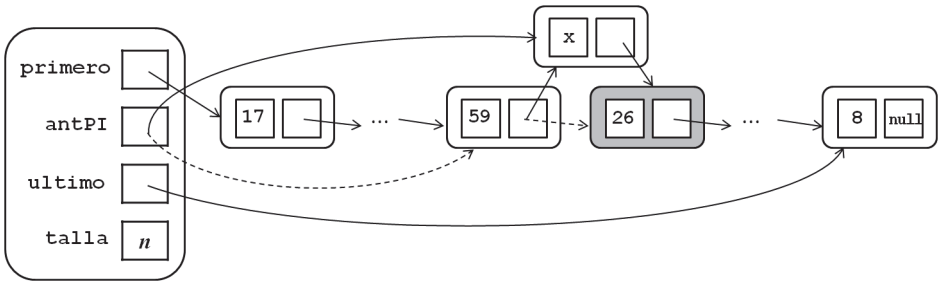
Figura 16.33: Esquema gráfico de una lista con punto de interés vacía con `primero, antPI, ultimo` y nodo ficticio.

Nótese que usar el nodo ficticio sirve sobre todo para simplificar el código de las operaciones, pues cualquier inserción o eliminación de un nuevo nodo se hará siempre después de un nodo preexistente, incluso cuando el elemento insertado o eliminado sea el primero de la lista. Por ejemplo, para la operación de inserción, en la figura 16.34 se muestran los dos casos que se deberían considerar en una lista con punto de interés con `primero`, `antPI`, `ultimo` y sin nodo ficticio: el punto de interés está en el inicio (figura 16.34(a)) o en cualquier otra posición (figura 16.34(b)).

[2]El dato del nodo ficticio, puesto que no es un dato real de la lista, se inicializa a un valor especial que resulte fácil de identificar como, por ejemplo, `Integer.MIN_VALUE`.



(a) Inserción en cabeza.



(b) Inserción en cualquier otra posición.

Figura 16.34: Inserción en una lista con punto de interés con `primero, antPI, ultimo` y sin nodo ficticio.

El código correspondiente debería ser:

```
public void insertar(int x) {
    if (antPI == null) {
        primero = new NodoInt(x, primero);
        antPI = primero;
    else {
        antPI.siguiente = new NodoInt(x, antPI.siguiente);
        antPI = antPI.siguiente;
    }
    if (antPI.siguiente == null) { ultimo = antPI; }
    talla++;
}
```

Con el uso del nodo ficticio, el código de dicha operación se simplifica al no tener que comprobar si el punto de interés es o no el primer nodo de la lista, tal como se observa en la figura 16.35.

En la figura 16.36 se puede ver la implementación de parte de la clase `ListaPIIntEnlaF`, incluyendo el método de inserción. Se deja como ejercicio completar el resto de métodos de esta clase.

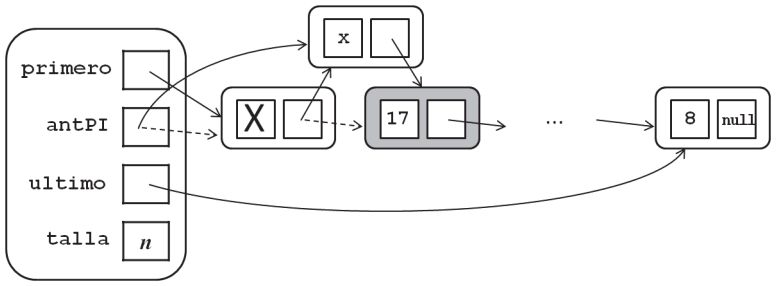


Figura 16.35: Inserción en una lista con punto de interés con `primero, antPI, ultimo` y nodo ficticio.

```
package lineales;
import java.util.NoSuchElementException;
/**
 *  Clase ListaPIIntEnlaF: Lista con punto de interés (o cursor) de int.
 *  Implementación enlazada con primero, antPI, ultimo y nodo ficticio.
 *  @author Libro IIP-PRG
 *  @version 2016
 */
public class ListaPIIntEnlaF {
    private NodoInt primero, antPI, ultimo;
    private int talla;

    /** Crea una ListaPIIntEnlaF vacía. */
    public ListaPIIntEnlaF() {
        primero = new NodoInt(0);
        antPI = primero; ultimo = primero; talla = 0;
    }

    /** Sitúa el cursor al inicio de la lista. */
    public void inicio() { antPI = primero; }

    /** Sitúa el cursor al final de la lista. */
    public void fin() { antPI = ultimo; }

    /** Comprueba si el cursor está al final de la lista.
     *  @return boolean, true si el cursor está al final
     *                   y false en caso contrario.
     */
    public boolean esFin() { return (antPI == ultimo); }

    /** Inserta x en la lista, delante del cursor.
     *  @param x int, el elemento a insertar.
     */
    public void insertar(int x) {
        NodoInt nuevo = new NodoInt(x, antPI.siguiente);
        antPI.siguiente = nuevo;
        if (ultimo == antPI) { ultimo = nuevo; }
        antPI = antPI.siguiente;
        talla++;
    }
```

Figura 16.36: Clase `ListaPIIntEnlaF` con `primero, antPI, ultimo` y nodo ficticio.

16.3 Problemas propuestos

1. Implementar una clase `NodoString` análoga a `NodoInt` excepto que sus datos deben ser de tipo `String`. Diseñar una clase `PalabrasOrd`, cuyos objetos tengan un atributo `sec` de la clase `NodoString` y un atributo entero `talla` que cuente el número de elementos en `sec`. Los elementos en `sec` se deben mantener ordenados ascendentemente según el orden de `String`.

   Teniendo esto en cuenta, hay que implementar los siguientes métodos en la clase `PalabrasOrd`:

   a) `public int talla()` que devuelva el número de elementos en la secuencia.

   b) `public void insertarOrd(String s)` que inserte ordenadamente `s` en la secuencia.

   c) `public boolean eliminar(String s)` que elimine la primera ocurrencia de `s` que aparezca en la secuencia. Si `s` no apareciera en la secuencia se debe devolver `false` y `true` en caso contrario.

   La clase se debe incluir en el mismo paquete que contenga la clase `NodoString`, para manejar explícitamente los enlaces de los nodos en la implementación de sus métodos.

2. Escribir una clase `OperacionesEnla` que implemente los siguientes métodos, utilizando explícitamente los enlaces de los nodos:

   a) `public static int maximo(NodoInt sec)` que, siendo `sec` una secuencia con al menos un nodo, busque el máximo elemento de `sec`.

   b) `public static int[] toArray(NodoInt sec)` que devuelva en un array del tamaño justo los elementos de la secuencia `sec`.

   c) `public static NodoInt moverADerecha(NodoInt sec)` que desplace todos los elementos de una secuencia `sec` una posición hacia la derecha. El último elemento deberá pasar a ser el primero. Por ejemplo, si la secuencia es `2, -23, 4, 12, 9, 55`, debe retornar `55, 2, -23, 4, 12, 9`.

   d) `public static NodoInt moverAIzq(NodoInt sec)` que desplace todos los elementos de una secuencia `sec` una posición hacia la izquierda. El primer elemento deberá pasar a ser el último. Por ejemplo, si la secuencia es `2, -23, 4, 12, 9, 55`, debe retornar `-23, 4, 12, 9, 55, 2`.

   e) `public static NodoInt invertir(NodoInt sec)` que invierta el orden de los elementos de una secuencia `sec` con un coste lineal. Por ejemplo, si la secuencia es `2, -23, 4, 12, 9, 55`, debe retornar `55, 9, 12, 4, -23, 2`.

f) `public NodoInt menoresQue(NodoInt sec,int e)` que, con un coste lineal, devuelva una secuencia enlazada con los elementos menores que `e` y en el mismo orden que aparecen en `sec`. Por ejemplo, si `sec` es `2, -23, 4, 12, 9, 55` y `e` es `10`, debe retornar `2, -23, 4, 9`. El coste deberá ser lineal con la longitud de `sec`.

La clase se debe escribir en un paquete que contenga `NodoInt`.

3. En un paquete que contenga la clase `NodoInt`, escribir una clase `OrdEnla` que, utilizando explícitamente los enlaces de los nodos, implemente los siguientes métodos:

   a) `public static NodoInt insDir(NodoInt sec)` que ordene `sec` mediante una estrategia de inserción directa, es decir, insertando ordenadamente los sucesivos datos de `sec` en la secuencia resultante.

   b) `private static int longitud(NodoInt sec)` que cuente y devuelva el número de elementos de `sec`.

   c) `private static NodoInt mezcla(NodoInt sec1,NodoInt sec2)` que, dadas dos secuencias enlazadas de enteros ordenadas ascendentemente, devuelva una tercera con la mezcla natural o fusión ordenada de ambas (sección 12.5.1). El coste ha de ser $\Theta(n)$, siendo n el número total de elementos a fusionar.

   d) `private static NodoInt mergesort(NodoInt sec,int n)` que ordene `sec` siguiendo una estrategia recursiva análoga a la del método de ordenación de arrays del mismo nombre (sección 12.4). El segundo parámetro ha de ser la longitud de `sec`.

   En el caso general del método, al partir la secuencia enlazada en dos subsecuencias de longitud similar, si el número de elementos es par, las dos subsecuencias tendrán la misma longitud; en caso contrario, la primera subsecuencia se quedará con un nodo más que la segunda. Una vez ordenadas las dos subsecuencias, se usará el método `mezcla` para completar la ordenación.

   e) `public static NodoInt mergesort(NodoInt sec)` que ordene `sec` usando el método privado del mismo nombre.

   Realizar el análisis de la complejidad de `insDir` y de `mergesort`.

4. Sobrescribir, en las clases `PilaIntArray` y `PilaIntEnla`, los métodos `toString()` y `equals(Object)` de la clase `Object` para adecuarlos a dichas clases. Estudiar su complejidad temporal.

5. Escribir un método recursivo y otro iterativo para invertir una pila.

6. Escribir un método para calcular la suma de los elementos de una pila de enteros.

7. Escribir un método recursivo que, dada una pila de enteros, obtenga una nueva pila con los mismos elementos que la original pero cambiados de signo.

8. Se dice que una pila `p` es *sombrero* de una pila `q` si todos los elementos de `p` están en `q` en el mismo orden y en posiciones más próximas a la cima. Escribir un método recursivo para comprobar si una pila `p` es *sombrero* de otra pila `q`.

9. Escribir un método recursivo para transformar una pila en una cola, de forma que el elemento situado en el tope de la pila quede el último en la cola.

10. Escribir un método para mostrar por pantalla el contenido de una cola.

11. La implementación de una cola mediante nodos enlazados (sección 16.2.2) contempla 2 atributos (`primero` y `ultimo`) para referenciar a los nodos que ocupan la primera y la última posición de la cola, respectivamente.

    Implementar una versión alternativa que sólo tenga 1 atributo referencia, en este caso al último nodo de la cola, mediante la definición de una secuencia circular, es decir, haciendo que el `siguiente` del `ultimo` nodo de la secuencia sea el primero de ésta, en lugar de `null` como es habitual. Hacerlo de modo que el coste de todas las operaciones permanezca en $\Theta(1)$.

12. Se supone una palabra representada, respectivamente, como una pila y como una cola de caracteres. Escribir para ambas representaciones un método `capicua(p)`, tal que devuelva `true` si `p` es una palabra capicúa y `false` en caso contrario. Comparar la complejidad del método en ambas representaciones.

13. La siguiente función invierte una cola:

```
public static void invertirCola(Cola c) {
    if (!c.esVacia()) {
        int x = c.desencolar();
        invertirCola(c);
        c.encolar(x);
    }
}
```

    Estudiar su complejidad temporal en los siguientes casos:

    - La implementación de la cola se ha realizado mediante una representación enlazada.
    - La implementación se ha realizado mediante una representación con arrays no circular.
    - La implementación se ha realizado mediante una representación con arrays circular.

    Comparar el coste del algoritmo en las tres representaciones propuestas.

14. Sobrescribir, en las clases `ListaPIIntArray` y `ListaPIIntEnla`, los métodos `toString()` y `equals(Object)` de la clase `Object` para adecuarlos a dichas clases. Estudiar su complejidad temporal.

15. Añadir las siguientes operaciones a la clase `ListaPIIntEnla`:

    a) `public void borrar(x)` que borre el elemento `x` de la lista. Si `x` aparece más de una vez, borra la primera aparición. Si `x` no se encuentra en la lista, no hace nada.

    b) `public void anterior()` que cambie el punto de interés a la posición anterior a la actual en la lista. La operación no está definida si el punto de interés es el primer elemento de la lista.

    c) `private boolean buscar(NodoInt ant,int x)` que busque la primera ocurrencia de `x` desde el nodo siguiente a `ant` en adelante; si lo encuentra, mueve el punto de interés al nodo que contiene a `x`.

    d) `public boolean buscarInicio(int x)` que, invocando al método privado `buscar` con los parámetros adecuados, indique si `x` aparece en la lista. Si `x` se encuentra, sitúa el punto de interés en la primera ocurrencia de `x`. Si no aparece, el punto de interés no se mueve.

    e) `public boolean buscarSiguiente(int x)` que, invocando al método privado `buscar` con los parámetros adecuados, indique si `x` aparece en la lista desde la posición del punto de interés inclusive en adelante. Si `x` aparece, avanza el punto de interés a `x`. Si no aparece, el punto de interés no se mueve.

    Estudiar su complejidad temporal.

16. Escribir un método que, dadas dos listas con punto de interés `l1` y `l2`, devuelva como resultado otra lista `l3` que contenga sólo aquellos elementos de `l1` que no estén en `l2` y los de `l2` que no estén en `l1`. Sólo se permite utilizar las operaciones definidas para el tipo.

Más información

[AGH01] K. Arnold, J. Gosling, and D. Holmes. *El lenguaje de programación Java*. Addison-Wesley, 2001. Capítulo 21.

[Eck15] D.J. Eck. *Introduction to Programming Using Java, Seventh Edition*. 2015. URL: `http://math.hws.edu/javanotes/`. Capítulo 9 (9.2 y 9.3).

[SM16] W.J. Savitch. *Absolute Java, Sixth Edition*. Pearson Education, 2016. Capítulo 15.

[Wei00] M.A. Weiss. *Estructuras de datos en Java: compatible con Java 2*. Addison-Wesley, 2000. Capítulos 6, 15 y 16.

[Wir82] N. Wirth. *Algoritmos + Estructuras de Datos = Programas*. Ediciones del Castillo, 1982. Capítulo 4 (4.1 y 4.2).