Manejo de listas encadenadas

Introducción

El encadenamiento de objetos mediante campos que sirven para referenciar de unos a otros permite crear estructuras complejas y flexibles, adaptadas a las características particulares de un problema concreto. Las listas encadenadas son estructuras encadenadas lineales formadas por una secuencia de nodos, cada uno de los cuales, salvo el último, tiene un único sucesor (Ilustración 1).

Ilustración 1 lista encadenada simple

Este documento ilustra las operaciones básicas de manejo de listas encadenadas, las que permiten añadir y quitar nodos de una lista, tomando como ejemplo, su aplicación en la implementación de contenedores dinámicos. Un contenedor es una clase cuyos objetos están diseñados para contener colecciones de objetos de otra clase. Los contenedores se diferencian entre sí por la forma en que se añaden o extraen elementos de la colección y por la forma en que se accede a los elementos almacenados.

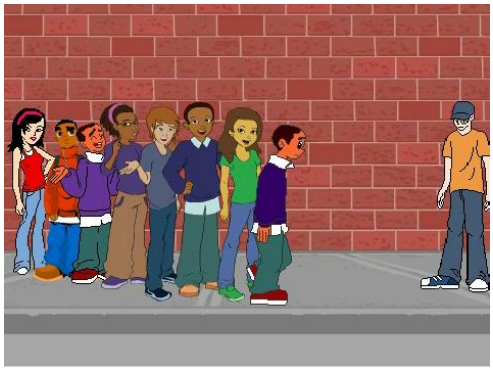

Ilustración 2 una cola

Ejemplo: una cola de números enteros

Consideremos, por ejemplo, una cola (Ilustración 2). Una cola es un contenedor que almacena una colección de elementos, normalmente a la espera de recibir un tratamiento, siguiendo la política "primero en entrar, primero en salir". Esto significa que el elemento que antes llega a la cola es el que

antes se trata y el que antes la abandona. Aparte de su utilidad, en general, son una clase de contenedor que se usa en muchos procesos informáticos; un ejemplo de ello son las colas de impresión, que se usan para regular el orden en que se tramitan las peticiones a una impresora.

Supongamos que queremos desarrollar una clase, *QueueOfInt*, que implemente la funcionalidad de una cola que almacene una colección de números enteros. Una opción es utilizar una *array* para almacenar los elementos, con sendos índices para indicar las posiciones del primer elemento (próximo elemento a tratar o sacar de la cola) y del último elemento (detrás del cual se añadirá el próximo que llegue).

```
public class QueueOfInt {
    private int[] data;     // Espacio para almacenar los elementos
    private int front = 0; // Posición del primer elemento
    private int rear = -1; // Posición del último elemento (-1 == vacía)

    /**
     * Crea una cola con capacidad para bound elementos
     * @param bound
     */
    public QueueOfInt(int bound) {
        data = new int[bound];
    }
    …
```

El inconveniente, como ya sabemos, es que ello nos obliga a establecer a priori una capacidad máxima para la cola y a hacer una reserva inicial de espacio acorde a dicha capacidad en el momento de crear un objeto de la clase *QueueOfInt*. La alternativa es usar una estructura encadenada.

Definición de los nodos de la estructura

Cuando queramos utilizar una estructura encadenada para almacenar una colección de elementos, el primer paso es definir cómo van a ser los nodos que actuarán como "ladrillos" de dicha estructura. Un nodo típico constará de dos partes: un conjunto de campos que representan la información almacenada en el nodo, y otro conjunto de campos de encadenamiento, que son referencias a otros nodos de la misma clase. El caso más sencillo, que vale para nuestro ejemplo, sería una lista encadenada simple, en la que cada nodo solo necesitaría un campo de encadenamiento. En nuestro ejemplo, también, se necesitaría solamente un campo de información, para representar un valor de tipo *int*.

```
public class QueueOfInt {
    private class Node { // Nodos que almacenan los elementos de la cola
        int info;        // Información almacenada en el nodo
        Node next;       // Encadenamiento al siguiente nodo
    }
```

…

La clase *Node* podría haberse declarado de manera independiente, pero se ha optado por declararla como una clase privada del propio contenedor (*QueueOfInt*), ya que es una opción práctica y bastante sencilla; aunque la alternativa de la declaración como clase independiente tendría la ventaja de permitir su reutilización en distintos contenedores.

La clase *Node* define cómo son los nodos que formarán la estructura. Para tener una estructura de elementos de tipo *Node* que almacene la información, declararemos dos variables para referenciar al primer y al último nodo de la misma:

```
public class QueueOfInt {

    …

    private Node front = null; // Referencia al primer elemento
    private Node rear = null;  // Referencia al último elemento
    …
```

Ahora, no nos hace falta tener un constructor como el que teníamos en la implementación con *arrays*, que servía para fijar el tamaño del *array* estableciendo la capacidad de la cola. Con una implementación basada en una lista encadenada en memoria no tenemos que fijar un límite a priori y, en principio, no hay ninguna otra cosa que haga falta inicializar.

Inserción de un elemento

La inserción de un nuevo elemento en la cola requiere la creación de un nuevo nodo y la modificación de algunos enlaces para engancharlo de forma adecuada a la lista de nodos.

```
    /**
     * Inserta element en la cola referenciada por this.
     * @param element
     */
    public void insert(int element) {
        if (rear == null) { // La cola está vacía
            rear = new Node(); // Se crea un nodo y rear y front
            front = rear;      // apuntan a él, que es el único
        } else { // La cola no está vacía
            rear.next = new Node(); // Se crea un nuevo nodo que
            rear = rear.next;       // se engancha al final y se
        }                           // actualiza rear

        rear.info = element; // Se almacena el elemento en el nuevo nodo
    }
```

Se distinguen dos casos particulares: que la estructura esté vacía, siendo, por tanto, la primera inserción que se va a realizar, o que no lo esté. Cuando la lista está vacía, tanto *front* como *rear* tendrán el valor *null*. En ese caso, sólo hay que crear un nuevo nodo, en el que se almacenará el valor correspondiente, y hacer que, tanto *rear* como *front*, lo referencien (Ilustración 3).

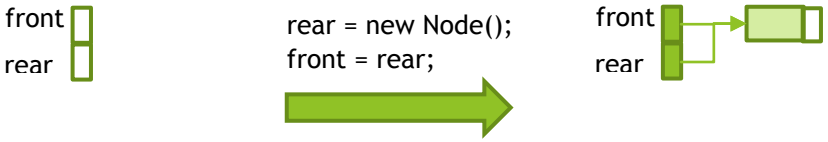

Ilustración 3 inserción en una lista vacía

Cuando la lista no está vacía, se crea un nuevo nodo que se engancha al final, haciendo que el campo *next* del último nodo (referenciado por *rear*) lo referencie. A continuación hay que actualizar *rear* para que pase a referenciar al nodo recién creado, que ahora es el último (Ilustración 4).

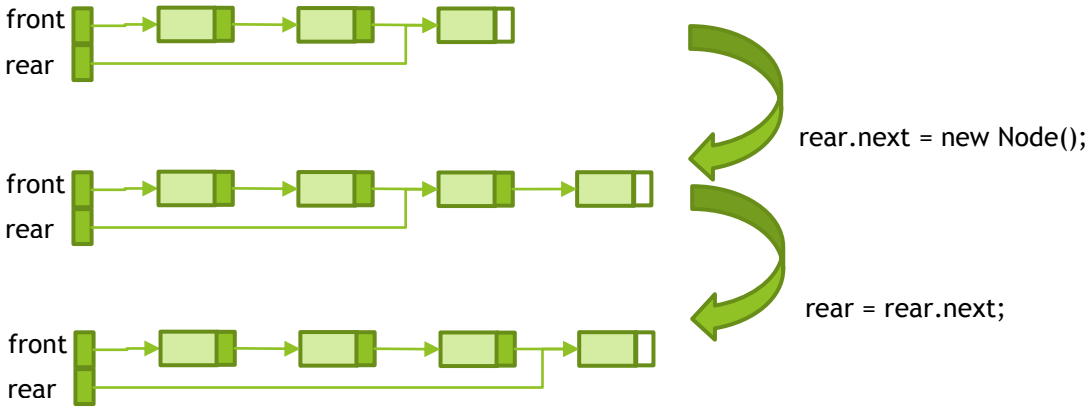

Ilustración 4 inserción al final de una lista

Extracción de un elemento

En una cola siempre se extrae el elemento que se encuentra en el frente de misma, que en la representación mediante lista encadenada es el que se encuentra en el primer nodo de la lista.

Si la cola no está vacía, simplemente, hacemos que la variable que referencia al primer nodo de la lista (*front*) pase a referenciar al segundo, que, de esta manera, se convierte en el primero. El que hasta ahora era el primer nodo queda sin referencia y será eliminado en su momento; en cualquier caso, ha dejado de formar parte accesible de la lista (Ilustración 5).

```
/**
 * Elimina el primer elemento de la cola
 */
public void remove() {
    if (front != null) { // La cola no está vacía
        front = front.next; // El segundo elemento pasa al frente

        if (front == null) { // Solo había un nodo, la cola
            rear = null;     // ha quedado vacía
        }
    }
}
```

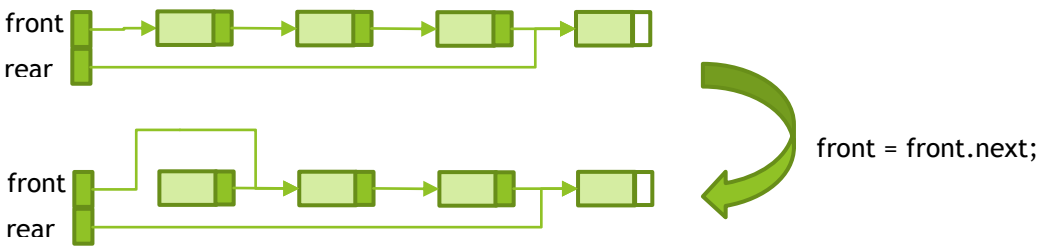

Ilustración 5 extracción al principio de una lista

En el caso particular de que solo haya un nodo en la lista (Ilustración 6), *front* tomará el valor *null* y habrá que hacer que la variable que referencia al último nodo (*rear*), que, en este caso, era el mismo nodo, tome también dicho valor. La variable *rear* no se modifica en ningún otro caso.

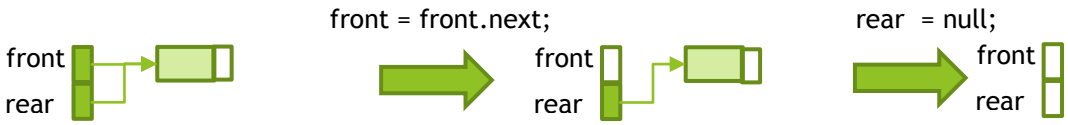

Ilustración 6 extracción al principio de una lista que tiene un solo nodo

Otras formas de insertar y extraer

La clase *QueueOfInt* nos ha servido para ilustrar los algoritmos de inserción de un nodo al final de una lista encadenada y de extracción del primer nodo de una lista encadenada, dado que son los que se aplican en el funcionamiento normal de una cola.

Tal como se comentó al principio, los contenedores se diferencian por la forma en la que se añaden y eliminan elementos.

En el caso de una implementación basada en listas encadenadas, las opciones de inserción y extracción de nodos de una lista, aparte de las ya vistas en el ejemplo de la clase *QueueOfInt* son:

- Insertar un nuevo nodo al principio de una lista
- Extraer el último nodo de una lista
- Insertar un nodo en una posición central de una lista
- Extraer un nodo en una posición central de una lista

Insertar un nodo al principio de una lista

Para ilustrar este caso, consideraremos la implementación de una pila, que es un contenedor en el que las inserciones y extracciones se hacen siguiendo la política "último en entrar, primero en salir", lo que, en una implementación mediante listas encadenadas se traduce en que, tanto las inserciones como las extracciones, se realizan por el comienzo de la lista.

La estructura para la implementación de una pila usando una lista encadenada sería muy similar a la usada para la implementación de una cola, con la salvedad de que no hace falta mantener una referencia al último nodo de la lista, ya que no se van a realizar operaciones en ese extremo.

```
public class StackOfInt {
    private class Node { // Nodos que almacenan los elementos de la pila
        int info;        // Información almacenada en el nodo
        Node next;       // Encadenamiento al siguiente nodo
    }

    private Node top = null; // Referencia al primer elemento
```

La inserción de un elemento como primer nodo de la lista es bastante sencilla:

```
    /**
     * Inserta element en la pila referenciada por this.
     * @param element
     */
    public void insert(int element) {
        Node newNode = new Node(); // Se crea un nuevo nodo
        newNode.info = element;    // Se pone el elemento en el nodo
        newNode.next = top;    // Se le hace apuntar al principio de la
                               // lista
        top = newNode;         // Se le convierte en el primer nodo de la
                               // lista
    }
```

Basta con crear un nuevo nodo, hacer que apunte como siguiente al que hasta ahora es el primero, y, a continuación hacer que la variable que referencia al primer nodo (*top*) pase a referenciar al nodo recién creado (Ilustración 7).

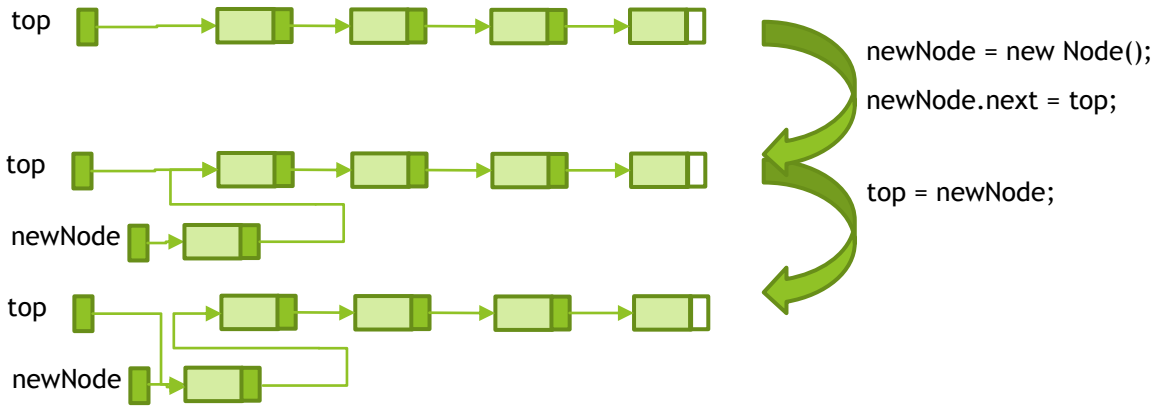

Ilustración 7 inserción al principio de una lista

No es necesario tener en cuenta el caso particular de que la lista esté vacía, dado que, en ese caso, *top* inicialmente tendrá el valor *null*, y la secuencia de asignaciones seguirá dando un resultado correcto.

La extracción del primer elemento de la pila sigue el mismo algoritmo ya mostrado para la cola, solo que, sin necesidad de preocuparse por actualizar la variable que referencia al último nodo, dado que no existe.

Extraer el último nodo de una lista

Este caso se puede dar, por ejemplo en una doble cola, que es un contenedor en el que las operaciones se realizan indistintamente por cualquiera de los dos extremos. Hay que señalar que el mecanismo usado en el caso de la cola, tener una variable que referencie al último nodo, resulta de gran ayuda para realizar inserciones al final. De no haberlo tenido, cada inserción requeriría recorrer previamente toda la lista para llegar al último nodo y realizar la inserción, con lo que las operaciones de inserción tardarían cada vez más, a medida que la lista fuese aumentando de tamaño.

Sin embargo, tener localizado el último nodo resulta inútil a la hora de realizar su extracción, ya que lo que se necesita es acceder al penúltimo para asignar a su enlace *next* el valor *null*. Con una estructura de lista encadenada simple siempre es necesario recorrer la lista para encontrar ese penúltimo nodo.

```
/**
 * Elimina el último elemento
 */
public void removeLast() {
    if (front != null) {  // La lista no está vacía
        Node prev = null; // Nodo previo
        Node act = front; // Nodo actual
        // Se avanza hasta alcanzar el último nodo
        while (act.next != null) {
            prev = act;
            act = act.next;
        }

        if (prev == null) { // Había solo un nodo
            front = null;
            rear = null;
        } else { // Se desengancha el último nodo
            prev.next = null;
            rear = prev;
        }
    }
}
```

Lo que hace el método *removeLast* es usar una variable, *act*, para recorrer la lista hasta llegar al último nodo, que se identifica porque su campo de encadenamiento tiene el valor *null*; y una segunda variable, *prev*, para referenciar durante el recorrido al predecesor del nodo referenciado por *act*. Cuando el recorrido acaba, *act* está referenciando al último nodo y *prev* al penúltimo. La variable *prev* se inicializa con el valor *null*, dado que el primer nodo no tiene predecesor. Si al terminar el recorrido *prev* sigue teniendo el valor *null* es que la lista sólo tenía un nodo; entonces hay que asignar el valor *null* a la variable que referencia el primer nodo, y no al campo *next* del penúltimo, que no existe.

Si la extracción del último nodo es una operación que se va a realizar con frecuencia, existe la alternativa de ganar tiempo, a costa de consumir algo más de espacio, utilizando una lista doblemente encadenada, en vez de una simple. Una lista doblemente encadenada (Ilustración 8) es una, en la que cada nodo tiene dos enlaces, uno que referencia a su sucesor y otro, que referencia a su predecesor.

```
private class Node { // Nodos que almacenan los elementos
    Node prev;       // Encadenamiento al nodo anterior
    int info;        // Información almacenada en el nodo
    Node next;       // Encadenamiento al siguiente nodo
}

private Node front = null; // Referencia al primer elemento
private Node rear = null;  // Referencia al último elemento
```

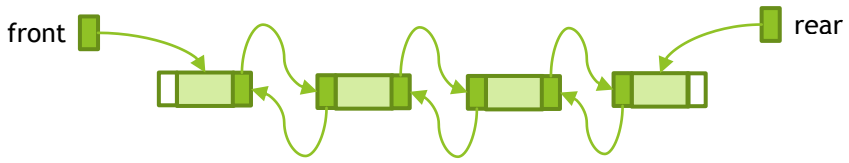

Ilustración 8 lista doblemente encadenada

El doble encadenamiento facilita notablemente la extracción del último nodo. El coste, aparte del espacio adicional, está en mantener actualizado el nuevo enlace en todas las operaciones.

```
/**
 * Elimina el último elemento
 */
public void removeLast() {
    if (front != null) { // La lista no está vacía
        if (rear.prev == null) { // Solo hay un nodo
            front = null;
            rear = null;
        } else {
            rear.prev.next = null; // Desengancha el último nodo
            rear = rear.prev; // El penúltimo se convierte en último
        }
    }
}
```

Insertar un elemento en una posición central de una lista

Para estudiar este caso vamos a suponer que tenemos una clase llamada *SortedSetOfInt* que almacena una colección ordenada y sin repetición de números enteros. Tanto la inserción, como la extracción de los elementos, la haremos en cualquier lugar de la lista, atendiendo al valor de los elementos. Si suponemos que hay la misma probabilidad de insertar en cualquier punto, entonces es innecesario mantener una referencia al último nodo, con lo que podríamos usar una estructura como la siguiente, en la que volvemos a la lista encadenada simple, aunque una lista doblemente encadenada podría tener sus ventajas, también:

```
public class SortedSetOfInt {
    private class Node{ // Nodos que almacenan los elementos de la lista
        int info;       // Información almacenada en el nodo
        Node next;      // Encadenamiento al siguiente nodo
    }

    private Node first = null; // Referencia al primer elemento
```

Con las premisas establecidas, la inserción de un elemento requiere la localización previa de la posición donde insertar, para proceder luego a actualizar los enlaces necesarios para enganchar un nodo con el nuevo elemento en esa posición de la lista.

```
/**
 * Inserta element en el SortedSetOfInt referenciado por this.
 * @param element
 */
public void insert(int element) {
    Node prev = null;
    Node act = first;

    // Recorre la lista buscando la posición de inserción
    while (act != null && act.info < element) {
        prev = act;
        act = act.next;
    }

    // Se insertará si el elemento no está ya en la lista
    if (act == null || act.info > element) {
        Node newNode = new Node(); // Se crea el nuevo nodo
        newNode.info = element;

        if (prev == null) { // El nuevo nodo va a ser el primero
            newNode.next = first;
            first = newNode;
        } else{ // El nuevo nodo va detrás del referenciado por prev
            newNode.next = prev.next;
            prev.next = newNode;
        }
    }
}
```

El método mostrado distingue el caso particular de que el nuevo nodo se inserte al principio de la lista, bien porque sea el primero que se inserta, bien porque tiene un valor menor que los que ya están almacenados. En ambos casos, no se entrará en el bucle que busca la posición de inserción, y la variable *prev* quedará con el valor *null*, indicando que el nuevo nodo no tendrá predecesor. En cualquier otro caso, se inserta detrás del nodo referenciado por *prev*, bien porque el siguiente, señalado por *act*, contiene un valor mayor que *element* (inserción en medio de la lista, Ilustración 9), o, porque todos los nodos tenían valores menores, *act* acabó el bucle con el valor *null*, y, *prev*, referencia al último, produciéndose una inserción por el final, aunque, la forma en que se realiza no se diferencia de una inserción en medio.

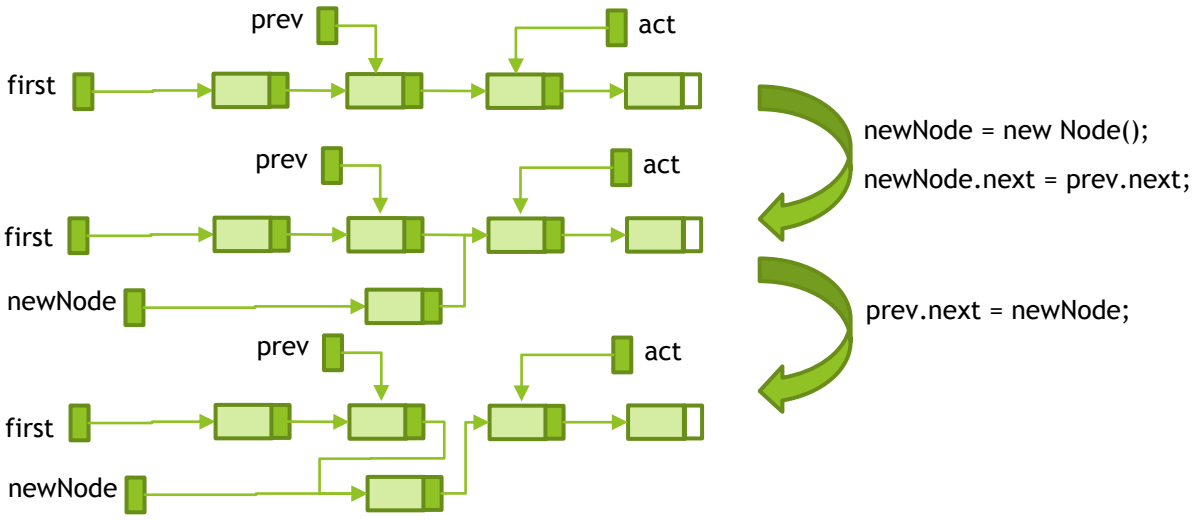

Ilustración 9 inserción en medio de una lista

La inserción ordenada también se puede afrontar de forma recursiva, basándonos en que, una lista encadenada se define recursivamente como:

- La lista vacía (referencia *null*)
- Un nodo seguido de una lista encadenada

```
/**
 * Inserta element en el SortedSetOfInt referenciado por this.
 * @param element
 */
public void insert(int element) {
    // Inmersión recursiva
    this.first = insert(this.first, element);
}

/**
 * Inserta element en la lista referenciada por act.
 * @param act
 * @param element
 */
private Node insert (Node act, int element) {
    if (act == null || act.info > element) {
        // element va justo en esta posición
        Node newNode = new Node();
        newNode.info = element;
        newNode.next = act;
        act = newNode;
```

```
        } else if (act != null && act.info < element) {
            // element va más adelante
            act.next = insert(act.next, element);
        }

        return act;
    }
```

Si nos fijamos, vemos que en la versión recursiva las instrucciones que realizan la inserción corresponden exactamente a las ya vistas para el caso de inserción al principio de una lista.

Extraer un nodo en una posición central de una lista

La extracción de un nodo de en medio de una lista la ilustraremos usando también la clase *SortedSetOfInt*, y suponiendo que queremos quitar un nodo que contenga un determinado valor. Al igual que en el caso de la inserción, la extracción en una posición central de una lista requiere localizar primero el nodo que se quiere extraer y arreglar luego el enlace del que lo precede para "puentearlo" hacia el siguiente, dejando el que queremos quitar inaccesible desde la lista (Ilustración 10).

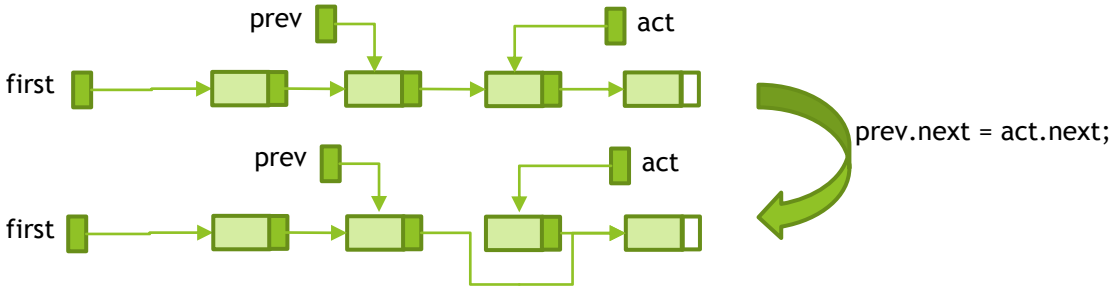

Ilustración 10 extracción en medio de una lista

Al igual que en el caso de la inserción, la extracción de un nodo en una posición central de una lista admite soluciones tanto recursivas como no recursivas. Mostramos en primer lugar una solución no recursiva que, como se puede ver, es estructuralmente muy parecida a la inserción mostrada en el apartado anterior:

```
/**
 * Elimina element del SortedSetOfInt referenciado por this
 * @param element
 */
public void remove2(int element) {
    Node prev = null;
    Node act = first;
```

```
        // Se busca element en la lista
        while (act != null && act.info < element) {
            prev = act;
            act = act.next;
        }

        if (act != null && act.info == element) { // Se encontró
            if (prev == null) { // Es el primer nodo
                first = act.next;
            } else { // No es el primer nodo
                prev.next = act.next;
            }
        }
    }
```

Mostramos también una solución recursiva. Hay que señalar que, tanto en la inserción como en la extracción, lo que hace la recursividad es el proceso de búsqueda; la inserción o la extracción en sí, se hacen en un caso base.

El método recursivo *remove* devuelve, bien la misma referencia que se le ha pasado (*act*), bien la referencia al nodo siguiente (*act.next*), esto último cuando el nodo referenciado por *act* tiene la información a eliminar. Al retornar, la referencia devuelta se asigna al campo next del nodo referenciado por *act* en la llamada actual, lo que, bien, lo deja como estaba, o bien, "puentea" el nodo a extraer, dependiendo de lo que haya devuelto la llamada con *act.next*.

```
    /**
     * Extrae el nodo con info element de la lista referenciada por act.
     * @param act
     * @param element
     * @return nodo pasado, si no es el que se quita, o el siguiente
     */
    private Node remove(Node act, int element) {
        if (act != null) { // Si la lista está vacía no se hace nada
            if (act.info == element) {
                act = act.next; // Se "salta" el nodo actual
            } else if (act.info < element) {
                // Se busca en el resto de la lista y se quita si está
                act.next = remove(act.next, element);
            } // Si act.info > element, element no está en la lista
        }

        return act;
    }
```

```java
/**
 * Elimina element del SortedSetOfInt referenciado por this
 * @param element
 */
public void remove(int element) {
    // Inmersión recursiva
    this.first = remove(this.first, element);
}
```

Conclusión

En este documento se han mostrado todas las opciones para insertar o extraer nodos de una lista simplemente encadenada (al principio, al final o en medio), incluyendo, en los casos pertinentes, tanto opciones recursivas como no recursivas. Además, para el caso de la extracción del último nodo, se ha incluido la opción de usar una lista doblemente encadenada, por la ventaja que aporta en este caso. La modificación del resto de los algoritmos para adaptarlos a listas doblemente encadenadas no entraña grandes dificultades; básicamente, hay que incluir código para mantener actualizado el enlace al nodo anterior en todo momento.

Todos los algoritmos se han mostrado en el contexto de la aplicación de las listas encadenadas como soporte para la implementación de diferentes contenedores. Se han usado para este propósito contenedores que almacenan colecciones de números enteros (*int*), aunque para las operaciones de inserción y extracción, el tipo de los datos almacenados es irrelevante (excepto que, para los contenedores ordenados como el *SortedSetOfInt*, deben poderse comparar).

Eso sí, ya sabemos que, al ser el tipo *int* un tipo primitivo, sus valores se almacenan directamente en los nodos; mientras que para datos de cualquier clase no primitiva lo que había en cada nodo sería una referencia a un objeto representado aparte (Ilustración 11), lo cual sigue siendo irrelevante a efectos de inserción y extracción, pero puede ser interesante tenerlo en cuenta al pensar en el diseño de estructuras encadenadas de cualquier género.

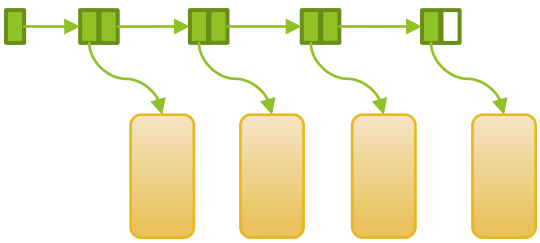

Ilustración 11