TEMA 3

Listas.

3.1. CONCEPTOS GENERALES.

Una lista es una estructura de datos lineal que se puede representar simbólicamente como un conjunto de nodos enlazados entre sí.

Las listas permiten modelar diversas entidades del mundo real como por ejemplo, los datos de los alumnos de un grupo académico, los datos del personal de una empresa, los programas informáticos almacenados en un disco magnético, etc.

La figura 3.1 muestra un ejemplo de lista correspondiente a los nombres y apellidos de un conjunto de alumnos con su código de matrícula.

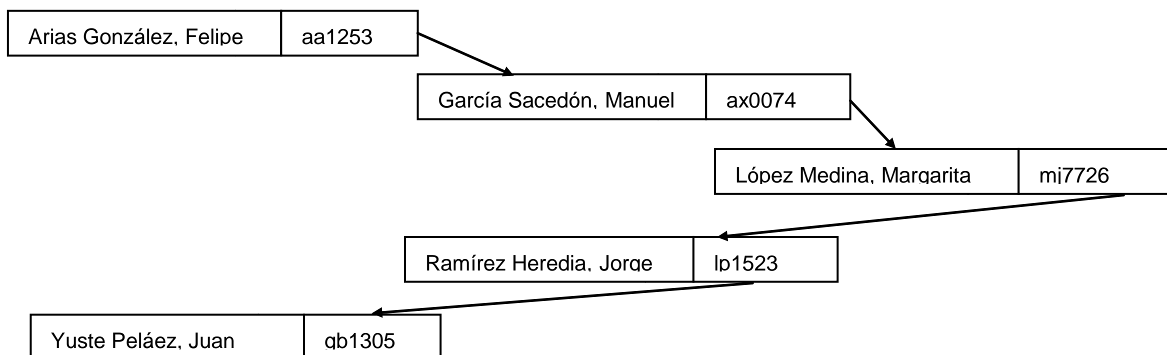

Figura 3.1. Ejemplo de Lista

Tal vez resulte conveniente identificar a los diferentes elementos de la lista (que normalmente estarán configurados como una estructura de registro) mediante uno de sus campos (clave) y en su caso, se almacenará la lista respetando un criterio de ordenación (ascendente o descendente) respecto al campo clave.

Una definición formal de lista es la siguiente:

"Una lista es una secuencia de elementos del mismo tipo, de cada uno de los cuales se puede decir cuál es su siguiente (en caso de existir)."

Existen dos criterios generales de calificación de listas:

- Por la forma de acceder a sus elementos.
  - Listas densas. Cuando la estructura que contiene la lista es la que determina la posición del siguiente elemento. La localización de un elemento de la lista es la siguiente:
    - Está en la posición 1 si no existe elemento anterior.
    - Está en la posición N si la localización del elemento anterior es (N-1).
  - Listas enlazadas: La localización de un elemento es:
    - Estará en la dirección k, si es el primer elemento, siendo k conocido.
    - Si no es el primer elemento de la lista, estará en una dirección, j, que está contenida en el elemento anterior.
- Por la información utilizada para acceder a sus elementos:
  - Listas ordinales. La posición de los elementos en la estructura la determina su orden de llegada.
  - Listas calificadas. Se accede a un elemento por un valor que coincide con el de un determinado campo, conocido como **clave**. Este tipo de listas se pueden clasificar a su vez en **ordenadas** o **no ordenadas** por el campo clave.

3.2. IMPLEMENTACIÓN DE LISTAS.

El concepto de lista puede implementarse en soportes informáticos de diferentes maneras.

- Mediante estructuras estáticas. Con toda seguridad resulta el mecanismo más intuitivo. Una simple *matriz* resuelve la idea (figura 3.2).

0	Arias González, Felipe	aa1253
1	García Sacedón, Manuel	ax0074
2	López Medina, Margarita	mj7726
3	Ramírez Heredia, Jorge	lp1523
4	Yuste Peláez, Juan	gb1305

Figura 3.2. Implementación de una lista densa mediante una estructura estática (matriz).

El problema de esta alternativa es el derivado de las operaciones de inserción y modificación.

En efecto, la declaración de una lista mediante una matriz implica conocer de antemano el número (o al menos el orden de magnitud) de elementos que va a almacenar, pudiendo darse las circunstancias de que si se declara pequeño podría desbordarse su capacidad o, en caso contrario, declararlo desproporcionadamente elevado provocaría un decremento de eficiencia.

Otro problema asociado es el tratamiento de los elementos eliminados. Dado que en el caso de no informar, de alguna manera, de la inexistencia de dicho elemento el nodo previamente ocupado (y ahora no válido) quedaría como no disponible.

Adicionalmente, si se desea trabajar con listas ordenadas el algoritmo de inserción debería alojar a los nuevos elementos en la posición o con la referencia adecuada.

Algunas soluciones, más o menos ingeniosas, permiten tratar estas circunstancias. La figura 3.3 muestra un ejemplo basado en una matriz de registros.

0	1	2	3	4	5	6	7	8
1	10	77	12	26	21	11	13	18
2	3	4	7	6	0	8	5	0

Figura 3.3. Ejemplo de implementación de lista enlazada mediante una estructura estática (matriz).

Otra posible representación de esta lista sería la siguiente:

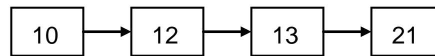

Figura 3.4. Lista enlazada correspondiente a la figura 3.3.

El tratamiento de las listas sobre matriz (tanto densas como enlazadas) se desarrollará más adelante, en el apartado 3.8.

- Mediante estructuras dinámicas.

Sin duda se trata de la mejor alternativa para la construcción de listas. Se trata de hacer uso de la correspondiente tecnología que implica el uso de referencias.

La idea consiste en declarar una referencia a un nodo que será el primero de la lista. Cada nodo de la lista (en la memoria dinámica) contendrá tanto la propia información del mismo como una referencia al nodo siguiente. Se deberá establecer un convenio para identificar el final de la lista[1].

La figura 3.5 muestra un ejemplo de implementación dinámica de la lista de la figura 3.4.

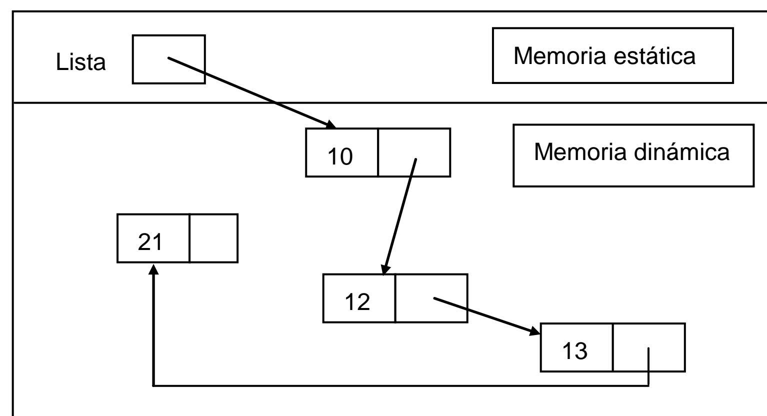

Figura 3.5. Implementación de una lista mediante estructuras dinámicas.

Lo explicado hasta el momento consiste en la solución más sencilla: cada nodo de la lista "apunta" al siguiente, con las excepciones del último elemento de la lista (su "apuntador, o "**referencia**" es un valor especial, por ejemplo *null*, en java) y del primero, que es "apuntado" por la referencia declarada en la memoria estática. Esta tecnología se identifica como "Listas enlazadas o unidireccionales". Existen otras posibilidades entre las que cabe mencionar: listas bidireccionales o doblemente enlazadas, listas circulares, listas con cabecera, listas con centinela o cualquier combinación de ellas.

---

[1] En java la referencia final tomará el valor null.

3.3. TRATAMIENTO DE LISTAS EN JAVA

Para la utilización de listas es necesario definir la clase *NodoLista* utilizando la siguiente sintaxis:

```
class NodoLista {
   public int clave;
   public NodoLista sig;
   public NodoLista(int x, NodoLista n) {
      clave = x;
      sig = n;
   }
}
```

Así como la clase Lista:

```
public class Lista {
   public Lista (String nombreLista) {
      inicio = null;
      nombre = nombreLista;
   }
   public NodoLista inicio;
   public String nombre;
}
```

La representación gráfica de una variable de la clase Lista llamada *lista1* que contenga los elementos 10, 13 y 21 sería:

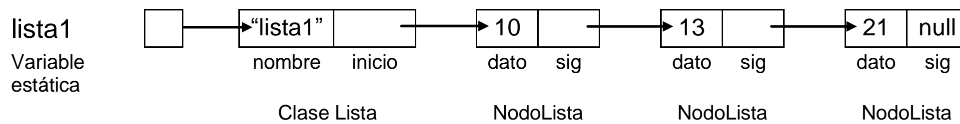

Figura 3.6. Ejemplo de lista enlazada.

3.4. ALGORITMOS BÁSICOS CON LISTAS[2].

Los algoritmos que implican el recorrido, parcial o total, de la lista pueden implementarse tanto de forma recursiva como iterativa.

3.4.1. Recorrido completo.

El recorrido de una lista de manera recursiva se puede realizar por medio de un método *static* invocado desde un programa que utilice la clase Lista. Esto implica utilizar el tipo *NodoLista* para avanzar por la lista y ver el contenido del campo clave. Merecen una consideración especial:

- La llamada inicial donde *lista* es el valor de la variable estática.
- El final del recorrido, que se alcanza cuando la lista está vacía.

En el siguiente método estático (*escribirLista*), se recorre una lista (*nodoLista*) mostrando en la pantalla el contenido de sus campos clave. Se utiliza un método de llamada (*escribirListaCompleta*), que recibe como argumento un objeto de la clase *Lista* (*lista*):

```
static void escribirLista (NodoLista nodolista) {
   if (nodoLista != null) {
      System.out.print (nodoLista.clave + " ");
      escribirLista (nodoLista.sig);
    }
   else System.out.println (" FIN");
 }
static void escribirListaCompleta (Lista lista) {
   if (lista != null) {
      System.out.print (lista.nombre + ": ");
      escribirLista (lista.inicio);
   }
   else System.out.println ("Lista vacía);
}
```

Si se aplica el algoritmo anterior a la lista de la figura 3.6, el resultado sería la secuencia:

lista1: 10 13 21 FIN.

La ejecución de *escribirListaCompleta* implicaría una llamada inicial al método *escribirLista,* pasando como argumento *lista.inicio* (la referencia al nodo de clave 10) y 3 llamadas recursivas al método *escribirLista*:

---

[2] Los siguientes algoritmos se han desarrollado considerando implementaciones de la lista como estructuras dinámicas. A efectos de la realización de prácticas se utiliza la sintaxis del lenguaje java.

- En la primera llamada recursiva, *nodoLista* es el contenido del campo *sig* del nodo de clave 10. Es decir, una referencia al nodo de clave 13.

- En la segunda, *nodoLista* es el contenido del campo *sig* del nodo de clave 13. Es decir, una referencia al nodo de clave 21.

- En la tercera, *nodoLista* es el contenido del campo *sig* del nodo de clave 21, es decir, *null*. Cuando se ejecuta esta tercera llamada se cumple la condición de finalización y, en consecuencia, se inicia el proceso de "vuelta". Ahora *nodoLista* toma sucesivamente los valores:

    - Referencia al nodo de clave 21 (campo *sig* del nodo de clave 13).
    - Referencia al nodo de clave 13 (campo *sig* del nodo de clave 10).
    - Referencia al nodo de clave 10 (el primer elemento de la lista).

El recorrido de una lista es una operación necesaria en los algoritmos de eliminación y, en muchos casos, también en los de inserción. La condición de finalización "pesimista" consiste en alcanzar el final de la lista (*nodoLista == null*). No obstante, normalmente se produce una terminación anticipada que se implementa *no realizando nuevas llamadas recursivas*.

En los siguientes apartados se van a presentar los diferentes tipos de listas, sobre las cuales se explican algunos algoritmos básicos: inicialización, búsqueda, inserción y eliminación.

3.5. LISTAS ORDINALES.

En las listas ordinales el orden dentro de la estructura lo establece la llegada a la misma. A diferencia de las listas calificadas, en este tipo de listas no existe ningún elemento que identifique el nodo, y por lo tanto, los valores se pueden repetir. El criterio de inserción resulta específico en cada caso (se podría insertar por el principio, o bien por el final). Veremos a continuación dos ejemplos de listas ordinales que ya hemos tratado como TADs: las pilas y las colas.

3.5.1. Pilas.

Como se ha visto en el tema 2, una pila es una agrupación de elementos de determinada naturaleza o tipo (datos de personas, números, procesos informáticos, automóviles, etc.) entre los que existe definida una relación de orden (**estructura de datos**). En función del tiempo, algunos elementos de dicha naturaleza pueden llegar a la pila o salir de ella (**operaciones / acciones**). En consecuencia el estado de la pila varía.

En una pila (comportamiento LIFO -*Last In First Out*-) se establece el criterio de ordenación en sentido inverso al orden de llegada. Así pues, el último elemento que llegó al conjunto será el primero en salir del mismo, y así sucesivamente. Las figuras 2.12 y 2.13 ilustran respectivamente el concepto de pila de números enteros y su implementación mediante una lista dinámica.

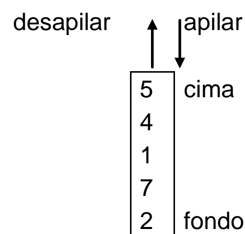

Figura 3.7. Modelo gráfico de Pila

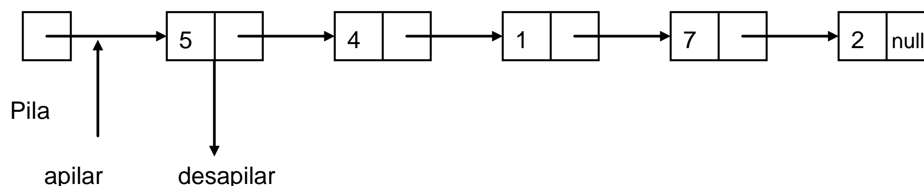

Figura 3.8. Implementación de una pila mediante una lista dinámica

La estructura de datos de la pila y el constructor utilizado sería:

```
package tadPila;
//En esta clase se define el nodo:
class NodoPila {
// Constructor
   NodoPila (int elemento, NodoPila n) {
      dato = elemento;
      siguiente = n;
   }
// Atributos accesibles desde otras rutinas del paquete
   int dato;
   NodoPila siguiente;
}
```

Y la interfaz utilizada sería la que se ha visto en el tema de TADs:

```
package tadPila;
import java.io.*;
public interface Pila {
   void inicializarPila ();
   boolean pilaVacia ();
   void eliminarPila ();
   int cima () throws PilaVacia;
   void apilar (int x);
   int desapilar () throws PilaVacia;
   void decapitar () throws PilaVacia;
   void imprimirPila ();
   void leerPila () throws NumberFormatException, IOException;
   int numElemPila ();
}
```

A continuación se muestra la clase TadPila, con el constructor y los algoritmos correspondientes a las operaciones *pilaVacia, inicializarPila*, así como *apilar* y *desapilar* que operan al principio de la lista por razones de eficiencia.

```
package tadPila;
public class TadPila implements Pila {
   protected NodoPila pila;
   public TadPila () {
      pila = null;
   }
   public void inicializarPila() {
      pila = null;
   }
   public boolean pilaVacia () {
      return pila == null;
   }
   public void apilar (int dato) {
      NodoPila aux = new NodoPila (dato,pila);
      pila = aux;
   }
```

```
    public int desapilar () throws PilaVacia {
        int resultado;
        if (pilaVacia ())
            throw new PilaVacia ("Desapilar: La pila está vacía");
        resultado = pila.dato;
        pila = pila.siguiente;
        return resultado;
    }
}
```

3.5.2. Colas.

Como se ha visto en el tema 2, una cola es una agrupación de elementos de determinada naturaleza o tipo (datos de personas, números, procesos informáticos, automóviles, etc.) entre los que existe definida una relación de orden. En función del tiempo, pueden llegar a la cola o salir de ella algunos elementos de dicha naturaleza (**operaciones/acciones**). En consecuencia el estado de la cola varía.

En una cola (comportamiento FIFO -*First In First Out*-) se respeta como criterio de ordenación el momento de la llegada: el primero de la cola, será el que primero llegó a ella y, en consecuencia, el primero que saldrá, y así sucesivamente. Las figuras 2.14 y 2.15 ilustran respectivamente el concepto de cola de números enteros y su implementación mediante una lista dinámica.

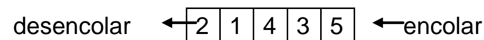

Figura 3.9. Modelo gráfico de Cola

Debido al comportamiento de la cola, la forma más eficiente de implementarla por medio de listas enlazadas, es utilizando dos referencias: una al principio de la cola (por donde desencolaremos) y otra al final (por donde encolaremos):

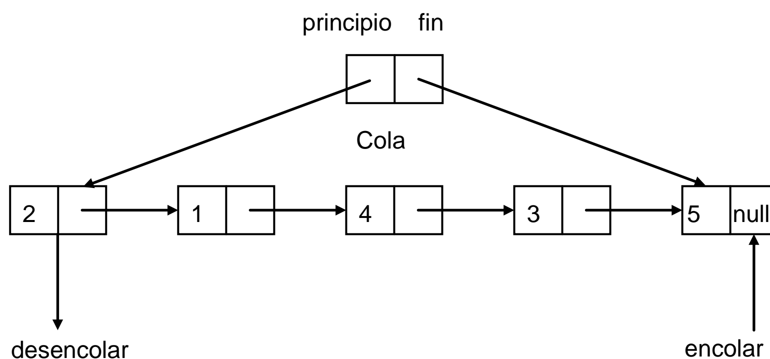

Figura 3.10. Implementación de una cola mediante una lista dinámica

La estructura de datos de la cola y el constructor sería:

```
package tadCola;
//En esta clase se define el nodo:
class NodoCola {
// Constructor
   NodoCola (int elemento, NodoCola n) {
      dato = elemento;
      siguiente = n;
   }
// Atributos accesibles desde otras rutinas del paquete
   int dato;
   NodoCola siguiente;
}
```

Y la interfaz utilizada sería la que se ha visto en el tema de TADs:

```
package tadCola;
import java.io.IOException;
public interface Cola {
   void inicializarCola ();
   boolean colaVacia ();
   void eliminarCola ();
   int primero ()throws ColaVacia;
   void encolar (int x);
   int desencolar () throws ColaVacia
   void quitarPrimero ()throws ColaVacia;
   void mostrarEstadoCola ();
   void imprimirCola ();
   void leerCola () throws NumberFormatException, IOException;
   int numElemCola ();
   void invertirCola ()throws ColaVacia;
}
```

A continuación se muestrala clase TadCola, que contiene el constructor, así como los algoritmos correspondientes a las operaciones *inicializarCola*, *colaVacia*, *desencolar* (opera al principio de la lista verificando la situación de excepción de recibir una cola vacía), *encolar* (opera al final de la lista)[3], *numElemCola* e *invertirCola*.

```
public class TadCola implements Cola {
   private NodoCola principio;
   private NodoCola fin;
   public TadCola () {
      principio = null;
      fin = null;
   }
   public void inicializarCola () {
      principio = null;
      fin = null;
   }
   public boolean colaVacia () {
      return principio==null;
   }
   public void encolar (int x) {
      NodoCola aux = new NodoCola(x,null);
      if (principio == null) {
         principio = aux;
         fin = aux;
      }
      else {
         fin.siguiente = aux;
         fin = aux;
      }
   }
   public int desencolar () throws ColaVacia {
      int resultado;
      if (colaVacia ())
         throw new ColaVacia ("Desencolar: La cola está vacía");
      resultado = principio.dato;
      principio = principio.siguiente;
      if (principio == null)
         fin = null;
      return resultado;
   }
   public int numElemCola () {
      NodoCola aux;
      int resul;
      aux = principio;
      resul = 0;
      while (aux != null) {
         ++resul;
         aux = aux.siguiente;
      }
      return resul;
   }
   public void invertirCola ()throws ColaVacia {
      int elem;
      if (!colaVacia ()) {
         elem = desencolar ();
         invertirCola ();
         encolar (elem);
      }
   }
}
```

[3] Se propone otra solución basada en una lista circular (apartado 3.7.1.) en la que la referencia a la lista es la del último nodo. De esta forma también se tiene acceso a ambos extremos (*cola*: último y *cola.sig*: primero).

3.6. LISTAS CALIFICADAS.

Se caracterizan por la existencia de un campo que identifica de manera unívoca cada uno de los nodos de la lista (*identificativo* o *clave*); lógicamente dicho valor debe ser único. Consideraremos dos casos: listas calificadas ordenadas y listas calificadas no ordenadas, en función de que la organización física de la lista se establezca siguiendo un criterio de ordenación (ascendente o descendente) o no de la clave. En ambos casos, no se permitirá la inserción de elementos con la clave repetida.

A lo largo de los siguientes apartados utilizaremos una lista calificada, cuyos nodos estarán compuestos por una clave y la referencia al siguiente nodo, utilizando la siguiente estructura:

```
public class NodoLista {
   public int clave;
   public NodoLista sig;
   public NodoLista (int x, NodoLista n) {
      clave = x;
      sig = n;
   }
}
```

La clase Lista incluirá las variables miembro y el constructor que aparecen a continuación:

```
public class Lista {
   public NodoLista inicio;
   public String nombre;
   public Lista (String nombreLista) {
      inicio = null;
      nombre = nombreLista;
   }
}
```

3.6.1. Listas calificadas no ordenadas.

En los siguientes apartados, utilizaremos una lista calificada, con sus elementos sin ordenar. La inicialización y el recorrido completo serían similares a los vistos en el apartado 3.2. (algoritmos básicos con listas), cambiando los tipos de datos. En los siguientes apartados, veremos como se buscan, añaden y borran elementos de una lista calificada no ordenada.

3.6.1.1. Búsqueda.

Se trata de localizar una clave de valor determinado (pasado como argumento) sin entrar en consideraciones de qué se va a hacer con ella. En cualquier caso este tipo de algoritmos no tienen efecto alguno sobre la estructura (no se modifica el número de nodos).

En principio la condición de finalización se consigue al llegar al final de la lista (*inicio == null*). Se produce una terminación anticipada en caso de encontrar la clave buscada.

El siguiente algoritmo es un ejemplo de método booleano de objeto (*busqueda*) que recibiendo como argumento un dato (*elem*) devuelve *true* en caso de encontrar el elemento, y *false* si el elemento no está en la lista.

```
static boolean busqueda (NodoLista nodoLista, int x) {
   boolean resul = false;
   if (nodoLista != null)
      if (nodoLista.clave == x)
         resul = true;
      else resul = busqueda (nodoLista.sig, x);
   return resul;
}
public boolean busqueda (int x) {
   return busqueda (inicio, x);
 }
```

3.6.1.2.Inserción.

El efecto de este tipo de algoritmos es incrementar el número de nodos. Para crear un nuevo nodo es necesario conocer tanto la naturaleza de la estructura utilizada (estática o dinámica) como los recursos del lenguaje de programación empleado. Por ejemplo, en el caso de las estructuras dinámicas en java el método es el siguiente:

```
NodoLista aux = new NodoLista (dato, null);
```

De esta forma se crea un nuevo nodo (apuntado por *aux*) cuya clave contiene el *dato* a insertar. De momento el nuevo nodo no está enlazado en la *lista*. El punto de inserción dependerá del criterio que se establezca.

Lo más sencillo y eficiente es insertar el nuevo nodo al principio de la *lista*. El resultado será el ilustrado en la figura 3.7.

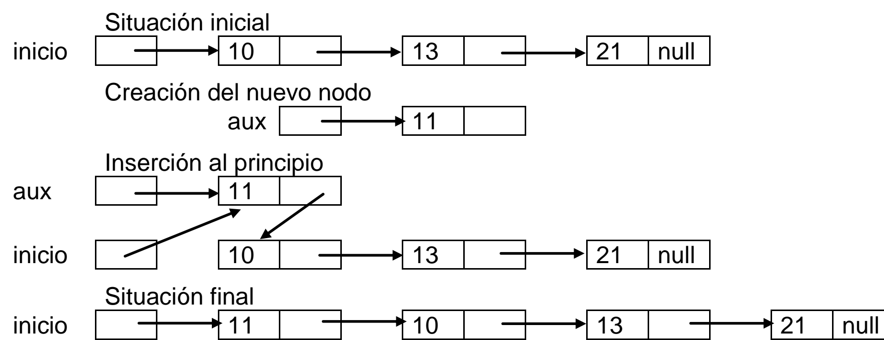

Figura 3.11. Inserción al principio de la lista.

El siguiente algoritmo muestra el método completo[4]:

```
public void insertarAlPrincipio (int x) {
   NodoLista aux = new NodoLista (x, inicio);
   inicio = aux;
}
```

El problema que tendría este algoritmo es que, dado que no comprueba si el elemento existe o no en la lista, podrá insertar claves repetidas. Para evitarlo, se podría realizar primero una búsqueda, y si la clave no estuviese ya en la lista (resultado del método *busqueda (dato) == true*), se insertaría. Otra posibilidad es realizar la inserción al final de la lista, comprobando si existe ya algún nodo con la clave buscada, utilizando un algoritmo recursivo.

Se utilizará un método de objeto de la clase *Lista*, que invoca a un método *static* al que se le pasa como argumento un *NodoLista*, y devuelve como resultado un *NodoLista*. En el método static vamos avanzando por la lista, haciendo las llamadas recursivas con *nodoLista.sig*. Si llegamos hasta el final de la lista sin localizar la clave buscada, se crea un nuevo nodo (*aux*), y lo devuelve como resultado.

Para enlazar correctamente el nodo, a la vuelta de la recursividad, se hace que *nodoLista.sig* apunte al mismo sitio que nos ha devuelto la llamada anterior como resultado.

```
static NodoLista insertarAlFinal (NodoLista nodoLista, int dato) {
   NodoLista aux = nodoLista;
   if (nodoLista!= null)
      if (nodoLista.clave != dato)
         nodoLista.sig = insertarAlFinal (nodoLista.sig, dato);
      else System.out.println ("la clave ya existe");
   else aux = new NodoLista (dato, nodoLista);
   return aux;
}
public void insertar (int dato) {
   inicio = insertarAlFinal (inicio, dato);
}
```

[4] Se contempla la situación excepcional de que la lista se encuentre inicialmente vacía.

La figura 3.8 ilustra el proceso seguido tras alcanzar la condición de finalización (*nodoLista == null*)[5].

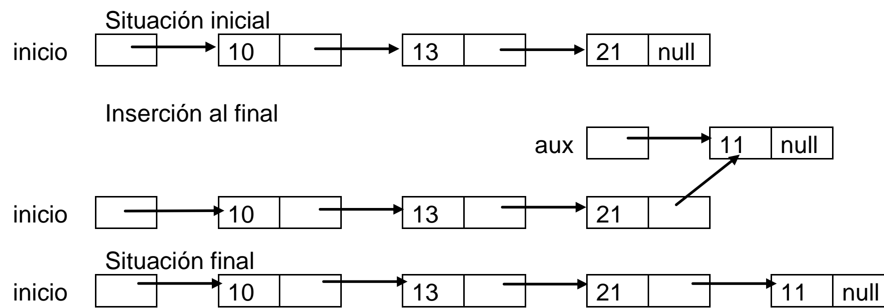

Figura 3.12. Inserción al final de la lista.

Si se desea realizar la inserción en una lista calificada no ordenada de manera iterativa, se podría utilizar el algoritmo que aparece a continuación. Obsérvese que, además de la variable *aux*, es necesario utilizar otros dos referencias, *actual* y *anterior* para recorrer la *lista* y verificar que el elemento no existe. Se utiliza, además, la variable *seguir*, de tipo *boolean*, que inicialmente es *true* y lo seguirá siendo a no ser que encontremos la clave en la *lista*.

```
public void insertarIterativo (int dato) {
   NodoLista anterior, actual, aux;
   boolean seguir;
   anterior = inicio;
   actual = inicio;
   seguir = true;
   while ((actual != null) && seguir)
      if (actual.clave == dato)
         seguir= false;
         else {
            anterior = actual;
            actual = actual.sig;
         }
   if (seguir) {
      aux = new NodoLista (dato, null);
      if (inicio == null)
         inicio = aux;
      else anterior.sig = aux;
      }
   else System.out.println ("Error. Elemento repetido");
}
```

Al realizar la inserción de manera iterativa, es necesario distinguir dos casos posibles: cuando insertamos un nodo en una lista vacía (tenemos que cambiar la referencia *inicio*), y cuando la lista ya tenía elementos previamente (insertamos al final, como nodo siguiente del elemento apuntado por *anterior*)

[5] Obsérvese que el algoritmo es válido cuando se recibe una lista inicialmente vacía.

3.6.1.3.Eliminación.

Este tipo de algoritmos reduce el número de nodos de la estructura. En la mayoría de los lenguajes de programación deberá prestarse atención especial a liberar el espacio de memoria de los nodos eliminados para posibles usos posteriores.[6]

Se procede a recorrer la lista comparando las sucesivas claves con el argumento recibido (*dato*).

La condición de finalización "pesimista" sería alcanzar el final de la lista, lo que significaría que no se habría encontrado la clave a eliminar. No obstante lo normal es que se produzca una terminación anticipada en el momento en que se encuentra la clave a eliminar.

La figura 3.9 ilustra gráficamente el mecanismo de eliminación.

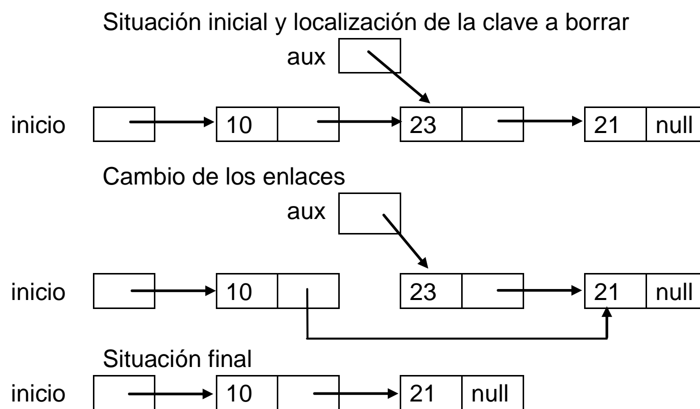

Figura 3.13. Eliminación de un nodo de la lista

El algoritmo utilizado es:

```
static NodoLista eliminar (NodoLista nodoLista, int dato) {
   NodoLista resul = nodoLista;
   if (nodoLista!= null)
      if (nodoLista.clave != dato)
         nodoLista.sig = eliminar (nodoLista.sig, dato);
      else resul = nodoLista.sig;
   else System.out.println ("la clave no existe");
   return resul;
}
public void eliminar (int dato) {
   inicio = eliminar (inicio, dato);
}
```

[6] En java no es necesario porque se realiza automáticamente, pero en los ejemplos se liberará la memoria no utilizada.

3.6.1.4. Listas Reorganizables.

Se denomina así al tipo de listas en las que la posición de los elementos va variando en función de los accesos que se hacen sobre la estructura. Cada vez que se accede a un nodo de la lista, éste pasa a convertirse en el primer elemento de la misma, desplazando al elemento que antes era el primero.

Normalmente este tipo de listas se emplean con la intención de mejorar la eficiencia de un proceso. Por ejemplo, cuando se piensa que existe cierta probabilidad de acceder con mayor frecuencia a determinadas claves, suele proporcionar buenos resultados reubicar los nodos a los que se haya accedido recientemente al principio de la lista, mientras que los que se consultan poco se desplazan a las posiciones finales de la lista. Lógicamente este tratamiento no tiene sentido para listas calificadas ordenadas.

Los métodos de inserción o borrado, serían los correspondientes a las listas enlazadas calificadas no ordenadas.

A continuación, se realizará la búsqueda de un elemento de la lista (pasamos la clave que estamos buscando, y recibimos como resultado *true* si hemos encontrado dicha clave, y *false* en caso contrario). El algoritmo se desarrolla en dos métodos (*reorganizar*, y *buscarYSaltar*), que utilizan la variable miembro *aux* de tipo *NodoLista*. El método *reorganizar* recibe como argumento la clave, y se la pasa a *buscarYSaltar*. Dicho método devolverá como resultado una referencia (*aux*) con la dirección del nodo de clave cuyo valor se busca (o *null* en caso de que no exista) a la vez que separa dicho nodo de la lista. Una vez conocida dicha referencia, si es distinta de *null*, se procede a insertar al principio de la lista el nodo apuntado por *aux*.

La figura 3.23 muestra una ilustración del proceso en el que se accede al nodo de clave 13.

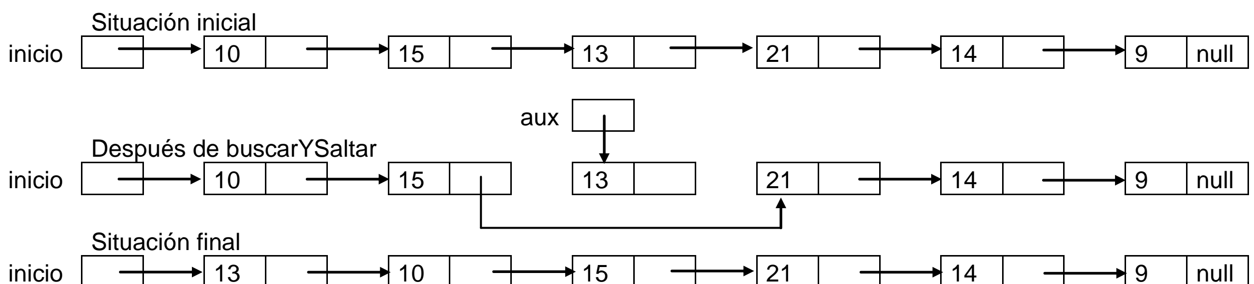

Figura 3.14. Lista reorganizable.

El método de búsqueda, con tratamiento recursivo, es equivalente al explicado para las listas calificadas no ordenadas, si bien incluye la variante de desenlazar de la lista el nodo cuya clave se busca para pasar su referencia al módulo principal.

En el código que aparece a continuación se muestra la solución descrita.

```
static NodoLista buscarYSaltar (NodoLista nodoLista, int dato) {
   NodoLista resul;
   if (nodoLista == null)
      resul = null;
   else if (nodoLista.clave != dato) {
         resul = buscarYSaltar (nodoLista.sig, dato);
         if ((nodoLista.sig == resul) && (resul != null))
            nodoLista.sig = resul.sig;
        }
        else resul = nodoLista;
   return resul;
}

static boolean reorganizar (Lista lista, int dato) {
   boolean resul = false;
   NodoLista aux;
   aux = buscarYSaltar (lista.inicio, dato);
   if (aux != null) {
      aux.sig = lista.inicio;
      lista.inicio= aux;
      resul = true;
   }
   return resul;
}
```

3.6.2. Listas calificadas ordenadas.

En los siguientes apartados, utilizaremos una lista calificada, con sus elementos ordenados ascendentemente. Los algoritmos de búsqueda, inserción y eliminación en listas calificadas ordenadas presentan las siguientes diferencias con respecto al caso de las listas no ordenadas:

- Los algoritmos de búsqueda y eliminación deben considerar como posibilidad adicional de finalización anticipada la situación de encontrar una clave superior al dato pasado como argumento. Esto se interpreta como una situación de error consistente en que la clave buscada no existe.
- En el caso de inserción la condición de finalización pesimista (*nodoLista == null*) se entiende como que la clave a insertar es de valor superior a la última de la lista. También es válido para insertar en una lista vacía. La terminación anticipada se produce cuando, como consecuencia de la exploración de la lista se encuentra una clave de valor igual o superior al de la clave a insertar.

A continuación se muestran ejemplos de dichos algoritmos.

3.6.2.1. Búsqueda.

Se trata de localizar una clave de valor determinado (pasado como argumento) sin entrar en consideraciones de qué se va a hacer con ella. En cualquier caso este tipo de algoritmos no tienen efecto alguno sobre la estructura (no se modifica el número de nodos).

En principio la condición de finalización se consigue al llegar al final de la lista (*inicio == null*). Se produce una terminación anticipada en caso de encontrar la clave buscada o una de valor superior a la misma.

El siguiente algoritmo es un método de objeto (*busqueda*) que, recibiendo como argumento un dato (*elem*) devuelve *true* en caso de encontrarlo, y *false* si el elemento no está en la lista. Llamamos a un método recursivo *static* (también llamado *busqueda*) al que le pasamos como argumento un *NodoLista* (*inicio).*

```
static boolean busqueda (NodoLista nodoLista, int x) {
   boolean resul = false;
   if (nodoLista != null)
      if (nodoLista.clave == x)
         resul = true;
      else if (nodoLista.clave < x)
            resul = busqueda (nodoLista.sig, x);
   return resul;
}
public boolean busqueda (int x) {
   return busqueda (inicio, x);
 }
```

3.6.2.2. Inserción.

En este caso se realiza un tratamiento recursivo recorriendo la *lista* y terminando anticipadamente en cuanto se accede a la primera clave de valor superior a la que se desea insertar (la condición de terminación general sería llegar al final de la lista, es decir, *nodoLista == null*). El nuevo nodo se deberá insertar en la posición anterior al nodo actual. La Figura 3.10 ilustra el proceso de inserción de un nodo de clave 11 en una lista en el momento de la finalización anticipada.

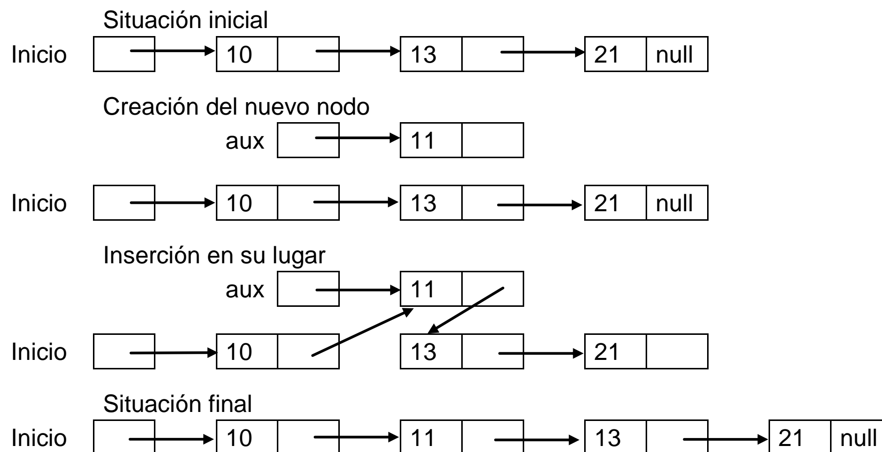

Figura 3.15. Inserción en una lista ordenada.

A continuación se muestra un algoritmo de inserción[7].

```
static NodoLista insertar (NodoLista nodoLista, int dato) {
   NodoLista resul = nodoLista;
   if (nodoLista != null)
      if (nodoLista.clave < dato)
         nodoLista.sig = insertar (nodoLista.sig, dato);
      else if (nodoLista.clave > dato)
         resul = new NodoLista (dato, nodoLista);
      else System.out.println ("la clave ya existe");
   else resul = new NodoLista (dato, nodoLista);
   return resul;
}
public void insertar (int dato) {
   inicio = insertar (inicio, dato);
}
```

Como en el caso de las listas calificadas no ordenadas, el método estático *insertar*, si añade el nuevo nodo, lo devuelve como resultado, para que a la vuelta se enlace el campo sig del nodo anterior con el nodo nuevo. Si todavía no hemos llegado a la posición de inserción, devolvemos como resultado el nodo actual.

[7] Obsérvese que el algoritmo es válido para insertar un elemento nuevo delante del primero, detrás del último y en una lista vacía. Así mismo, contempla el caso de intentar insertar una clave ya existente. Este último caso también produce una situación de terminación anticipada.

Para realizar la inserción se podría utilizar también un algoritmo iterativo. Como en el caso de las listas calificadas no ordenadas, tendríamos que utilizar las variables auxiliares *actual* y *anterior* para recorrer la lista. Además, utilizaríamos la variable *encontrado*, de tipo *boolean*, para salir del bucle de localización del hueco.

```
public void insertarIterativo (int dato) {
 NodoLista anterior, actual, aux;
 boolean encontrado;
 anterior = inicio;
 actual = inicio;
 encontrado = false;
 while ((actual != null) && !encontrado)
    if (actual.clave >= dato)
       encontrado = true;
    else {
       anterior = actual;
       actual = actual.sig;
      }
 if (actual == null) {
    aux = new NodoLista (dato, null);
    if (inicio == null)
       inicio = aux;
    else anterior.sig = aux;
  }
 else if (encontrado && (actual.clave > dato)) {
          aux = new NodoLista (dato, actual);
          if (inicio == actual)
             inicio = aux;
          else anterior.sig = aux;
         }
       else System.out.println ("Error. Elemento repetido");
 }
```

Al realizar la inserción en una lista calificada ordenada de manera iterativa, es necesario distinguir dos posibles casos: cuando insertamos un nodo al principio de la lista (tenemos que cambiar la referencia *inicio*), y cuando la lista ya tenía elementos previamente (insertamos o bien por la parte central de la lista, o bien al final, poniendo el nodo nuevo *aux* entre los elementos apuntados por *anterior* y *actual*)

3.6.2.3. Eliminación.

Para borrar un elemento, se procede a recorrer la lista comparando las sucesivas claves con el argumento pasado. La condición de finalización "pesimista" se alcanza al llegar al final de la lista, lo que significa que no se ha encontrado ni la clave a eliminar ni ninguna mayor. No obstante lo normal es que se produzca una terminación anticipada en el momento en que se encuentra o bien la clave a eliminar, o bien una clave de valor mayor que la buscada.

La figura 3.11 ilustra gráficamente el mecanismo de eliminación.

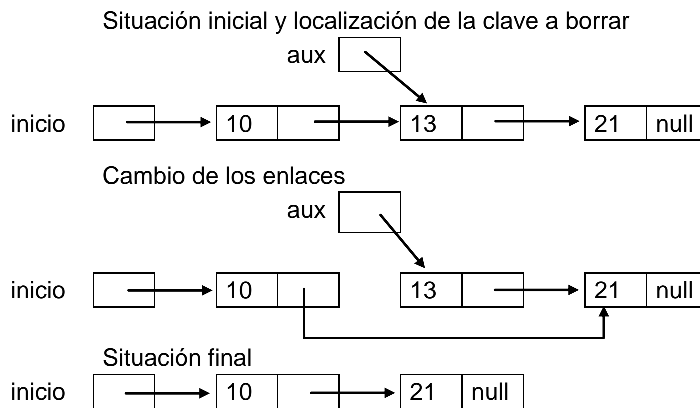

Figura 3.16. Eliminación de un nodo de una lista calificada ordenada.

El algoritmo utilizado es:

```
static NodoLista eliminar (NodoLista nodoLista, int dato) {
   NodoLista resul = nodoLista;
   if (nodoLista != null)
      if (nodoLista.clave < dato)
         nodoLista.sig = eliminar (nodoLista.sig, dato);
      else if (nodoLista.clave > dato)
            System.out.println ("la clave no existe");
      else resul = nodoLista.sig;
   else System.out.println ("la clave no existe");
   return resul;
}
public void eliminar (int dato) {
   inicio = eliminar (inicio, dato);
}
```

3.6.2.4. Mezcla de listas.

Se tratan a continuación dos casos en los que partiendo de dos listas (*lista1* y *lista2*) ordenadas de forma ascendente, se trata de obtener una nueva lista (*lista3*), con elementos de ambas. En el primer ejemplo crearemos *lista3* con los elementos comunes de *lista1* y *lista2* (intersección) en tanto que en el segundo *lista3* contendrá todos los elementos (sin repeticiones) de *lista1* y *lista2* (unión). En ambos casos realizaremos métodos *static* (pasando como argumentos objetos de la clase *NodoLista*).

3.6.2.4.1. Obtención de una lista con los elementos comunes de otras dos.

La condición de finalización se produce cuando se ha terminado de explorar alguna de las listas (*nodoLista1 == null || nodoLista2 == null*).

El avance recursivo por *nodoLista1*, *nodoLista2* o ambas tiene lugar como consecuencia del resultado de comparar los elementos actuales de ambas listas. En caso de que sean iguales, además, se produce la inserción del correspondiente nodo de *lista3*.

A continuación se muestra el código.

```
static NodoLista mezclaAnd(NodoLista nodo1, NodoLista nodo2, NodoLista nodo3){
   NodoLista resul;
   if (nodo1 != null && nodo2 != null)
      if (nodo1.clave < nodo2.clave)
         resul = mezclaAnd (nodo1.sig, nodo2, nodo3);
      else if (nodo1.clave > nodo2.clave)
            resul = mezclaAnd (nodo1, nodo2.sig, nodo3);
          else {
             nodo3 = mezclaAnd (nodo1.sig, nodo2.sig, nodo3);
             resul = new NodoLista (nodo1.clave, nodo3);
          }
   else resul = null;
   return resul;
}
static void mezclaAnd (Lista lista1, Lista lista2, Lista lista3) {
   lista3.inicio = mezclaAnd (lista1.inicio, lista2.inicio, lista3.inicio);
}
```

3.6.2.4.2. Obtención de una lista con todos los elementos de otras dos.

En este caso hay que recorrer ambas listas en su totalidad. La condición de finalización será ((*nodoLista1 == null) && (nodoLista2 == null)).*

Si una de las listas está vacía, o la otra tiene un elemento menor, lo insertaremos en la lista resultado y avanzaremos por esa lista. En caso de que ambas listas tengan el mismo elemento, lo insertaremos y avanzaremos por ambas a la vez.

- El proceso presenta tres situaciones (1+2) en función de que haya elementos en las dos listas o en una sí y en otra no:

- Cuando queden elementos en ambas listas (1 caso), compararemos su valor para insertar el menor de los elementos y avanzar por la lista con el elemento menor (o por ambas si son iguales),
- Cuando sólo queden elementos en una de las dos listas (2 casos), utilizaremos un método auxiliar (*copiar*), que copiará lo que quede en dicha lista sobre la lista resultado.

El algoritmo utilizado aparece a continuación:

```
   static NodoLista copiar (NodoLista nodoLista0) {
      NodoLista resul;
      if (nodoLista0 != null) {
         resul = copiar (nodoLista0.sig);
         resul = new NodoLista (nodoLista0.clave, resul);
      }
      else resul = null;
      return resul;
   }
   static NodoLista mezclaOr (NodoLista nodoLista1, NodoLista nodoLista2,
NodoLista nodoLista3) {
      NodoLista resul;
      if (nodoLista1 != null && nodoLista2 != null)
         if (nodoLista1.clave < nodoLista2.clave) {
            nodoLista3 = mezclaOr (nodoLista1.sig, nodoLista2, nodoLista3);
            resul = new NodoLista (nodoLista1.clave, nodoLista3);
         }
         else if (nodoLista1.clave > nodoLista2.clave) {
               nodoLista3 = mezclaOr (nodoLista1, nodoLista2.sig, nodoLista3);
               resul = new NodoLista (nodoLista2.clave, nodoLista3);
              }
              else {
                 nodoLista3 = mezclaOr (nodoLista1.sig, nodoLista2.sig, nodoLista3);
                 resul = new NodoLista (nodoLista1.clave, nodoLista3);
              }
      else if (nodoLista1 != null)
            resul = copiar (nodoLista1);
          else if (nodoLista2 != null)
                 resul = copiar (nodoLista2);
              else resul = null;
      return resul;
   }
   static void mezclaOr (Lista lista1, Lista lista2, Lista lista3) {
      lista3.inicio = mezclaOr (lista1.inicio, lista2.inicio, lista3.inicio);
   }
```

3.7. OTRAS IMPLEMENTACIONES.

El concepto de lista admite diferentes implementaciones con estructuras de datos tanto estáticas como dinámicas. A continuación se muestran, a título de ejemplo, dos posibles soluciones implementadas mediante estructuras dinámicas: listas circulares y bidireccionales. Así mismo, se plantea la técnica de realización de listas con cabecera y centinela, utilizada para las listas calificadas.

3.7.1. Listas circulares (anillos).

Se entiende por lista circular aquella en la que el último elemento de la lista tiene definido un enlace al primer elemento de la lista.

Desde un punto de vista físico una lista circular por propia naturaleza no tiene principio ni fin. No obstante, desde el punto de vista lógico, la referencia a la lista establece cómo determinar la finalización de su recorrido tanto iterativo como recursivo.

El nodo de referencia podría ser cualquiera de los nodos de la lista. A continuación se muestra como ejemplo (figura 3.16) el tratamiento de una lista calificada ordenada en que el nodo de referencia es el último (la clave de valor más alto). Este diseño facilita el acceso inmediato a ambos extremos de la lista lo que significaría un diseño eficiente, por ejemplo, para la implementación de colas[8] realizando la operación de encolar a continuación del nodo apuntado por *ultimo* (a la derecha en la figura) y desencolar en el extremo contrario (siguiente nodo).

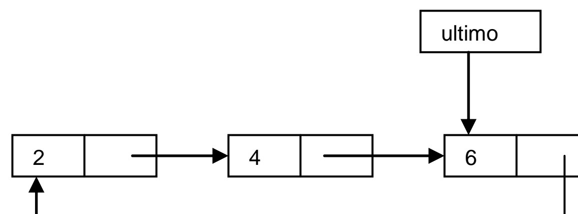

Figura 3.17. Ejemplo de lista circular

[8] Se sugiere al alumno como ejercicio.

3.7.1.1.Inserción.

Si se desea realizar la inserción en una lista calificada ordenada circular, hay que tener en cuenta tres casos diferentes:

- La lista está vacía (*ultimo == null*): deberemos crear un nodo y enlazarlo consigo mismo.
- Vamos a insertar al final, después del último elemento: tendremos que cambiar la referencia *ultimo*.
- El elemento se inserta en cualquier otra posición.

Para realizar la inserción de manera iterativa siguiendo los anteriores criterios, se puede utilizar el siguiente algoritmo, que crea el nodo y lo enlaza en su hueco correspondiente, una vez localizado el sitio donde se va a añadir el elemento:

```
public void insertar (int dato) {
   NodoLista aux, actual, anterior;
   if (ultimo == null) {
      aux = new NodoLista (dato);
      ultimo = aux;
      ultimo.sig = ultimo;
   }
   else {
      anterior = ultimo;
      actual = ultimo.sig;
      while ((actual.clave < dato) && (actual != ultimo)) {
         anterior = actual;
         actual = actual.sig;
      }
      if (actual.clave != dato) {
         aux = new NodoLista (dato);
         if ((actual != ultimo) || (actual.clave > dato)) {
            aux.sig = actual;
            anterior.sig = aux;
         }
         else if (actual.clave < dato) {
            aux.sig= actual.sig;
            actual.sig= aux;
            ultimo = aux;
         }
      }
      else System.out.println ("error, el elemento ya existe");
   }
}
```

3.7.1.2.Eliminación.

Si se va a realizar la eliminación de un nodo en una lista circular calificada ordenada de manera iterativa hay que contemplar, como en el caso de la inserción, tres casos diferentes:

- La lista tiene un solo nodo, que deseamos eliminar: deberemos borrar el nodo y apuntar la lista a *null*.
- Vamos a borrar el nodo final, (el que estamos apuntando con *ultimo*): tendremos que cambiar la referencia *ultimo*.
- El elemento se elimina de otra posición cualquiera.

Para realizar la eliminación de manera iterativa siguiendo los anteriores criterios, se puede utilizar el siguiente algoritmo:

```
public void eliminar (int x){
   NodoLista ant, act;
   if (ultimo != null) {
      ant = ultimo;
      act = ultimo.sig;
      while (act != ultimo && act.clave < x)  {
         ant = act;
         act = act.sig;
       }
      if (act.clave == x) {
         ant.sig = act.sig;
         if (ultimo == act)
            if (ultimo != ant)
               ultimo = ant;
            else ultimo = null;
      }
      else  System.out.println ("No existe el nodo de clave " + x);
   }
   else  System.out.println ("La lista está vacía ");
}
```

3.7.2. Listas bidireccionales (doblemente enlazadas).

Una lista se dice que es bidireccional o doblemente enlazada cuando, para cada uno de los elementos que la forman, existe una referencia al elemento anterior y otra al elemento siguiente dentro de la estructura. Gracias a esta estructura, se puede recorrer la lista en ambos sentidos.

En la implementación de este tipo de listas hay que considerar dos situaciones particulares:

- El elemento anterior al primero será el elemento nulo.
- El elemento siguiente al último será también el elemento nulo.

La figura 3.17 muestra un ejemplo de esta estructura:

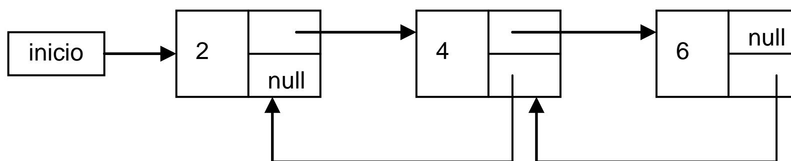

Figura 3.18. Ejemplo de lista bidireccional.

Para implementar una lista de estas características mediante una estructura de datos dinámica se puede utilizar la clase *NodoLista* modificada para incluir las dos referencias. La clase *NodoLista*, conteniendo las variables miembro y el constructor quedaría:

```
public class NodoLista {
   int clave;
   NodoLista sig, ant;
   public NodoLista (int x)  {
      clave = x;
      sig = null;
      ant = null;
   }
}
```

Para realizar la inicialización, búsqueda y recorrido completo de este tipo de listas utilizaríamos los mismos algoritmos que con las listas unidireccionales.

A continuación se muestran ejemplos de los algoritmos de inserción (*insertar*) y eliminación (*eliminar*) para una lista calificada ordenada[9] implementada mediante estructuras dinámicas bidireccionales.

[9] Se sugiere al alumno que realice los algoritmos correspondientes para listas calificadas no ordenadas.

3.7.2.1. Inserción.

Con carácter general la inserción se produce la primera vez que se encuentra un nodo cuya clave es de un valor superior al dato que se pretende insertar. Esto implica la modificación de cuatro referencias (dos en el nuevo nodo, otro en el que apunta al nodo de clave superior y otro del nodo siguiente que apunta al actual. La figura 3.18 ilustra el proceso (se supone que se intenta insertar un nodo de clave 5 en el momento de haber encontrado un nodo de clave superior, 6).

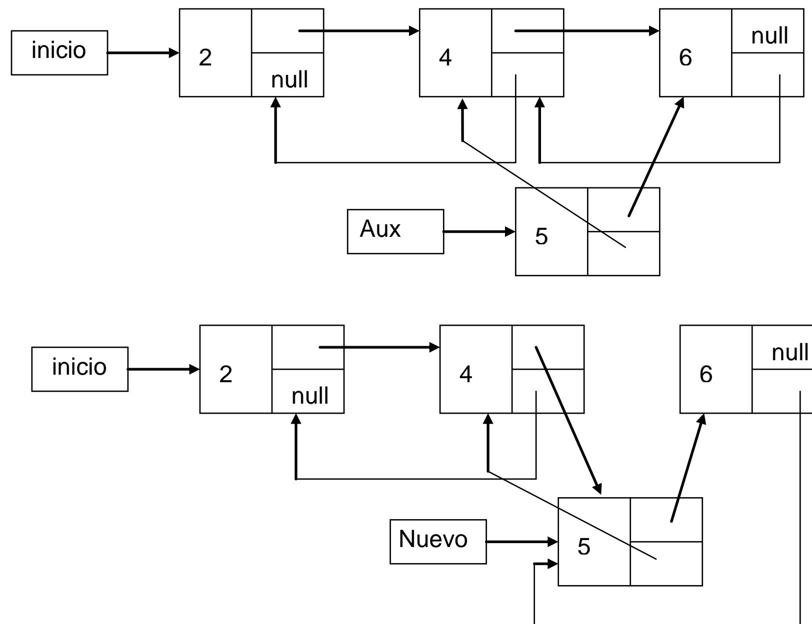

Figura 3.19. Simulación del módulo de inserción delante de un nodo

El código utilizado para insertar el nodo *aux* delante del nodo apuntado por *inicio* es el siguiente:

```
nuevo.sig = inicio;
nuevo.ant = inicio.ant;
inicio.ant = nuevo;
inicio = nuevo;
```

La terminación del proceso debe realizarse cuando se alcance el nodo cuyo campo *sig* sea *null (inicio.sig == null*) pues en caso de progresar más allá se perdería la referencia al nodo anterior. Si la clave que queremos insertar es superior a la última de la lista, habrá que insertarla en este momento, a la derecha del último nodo. El código siguiente indica como implementarlo y la figura 3.19 ilustra gráficamente el proceso.

```
nuevo.sig = null;
nuevo.ant = inicio;
inicio.sig = nuevo;
```

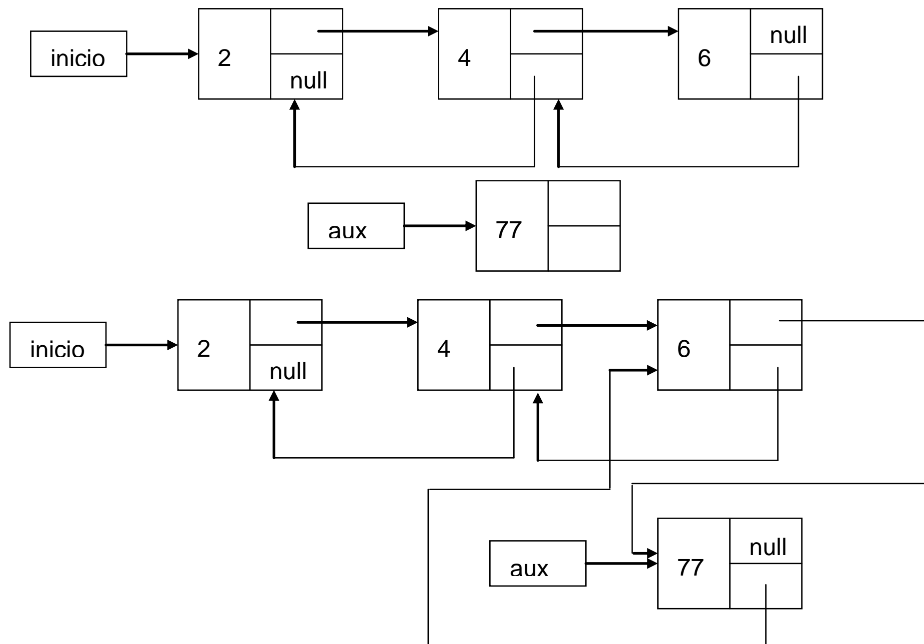

Figura 3.20. Inserción al final de la lista.

Para realizar la inserción en este tipo de listas de manera iterativa, se utilizan tres referencias auxiliares: *anterior* y *actual* para ir recorriendo la lista, y *nuevo* para generar el nodo. Además, necesitamos utilizar la variable booleana *encontrado*, para salir del bucle si localizamos el hueco donde debemos insertar el nodo.

El algoritmo utilizado aparece a continuación:

```
void insertar (int clave) {
   NodoLista anterior, actual, nuevo;
   boolean encontrado = false;
   anterior = inicio;
   actual = inicio;
   while ((actual != null) && !encontrado)
      if (actual.clave < clave) {
         anterior = actual;
         actual = actual.sig;
      }
      else encontrado = true;
   if (actual == null) {
      nuevo = new NodoLista (clave);
      if (inicio == null)
         inicio = nuevo;
      else {
         nuevo.ant = anterior;
         anterior.sig = nuevo;
      }
   }
```

```
    else if (actual.clave > clave) {
            nuevo = new NodoLista (clave);
            nuevo.sig = actual;
            nuevo.ant = actual.ant;
            actual.ant = nuevo;
            if (inicio != actual)
               anterior.sig = nuevo;
            else inicio  = nuevo;
          }
    else System.out.println ("error, la clave ya existe");
}
```

3.7.2.2.Eliminación.

Al realizar la eliminación de manera iterativa, de forma similar que en el caso de la inserción, se utilizan dos referencias auxiliares: *anterior* y *actual* para ir recorriendo la lista. Además, necesitamos utilizar la variable booleana *encontrado*, para salir del bucle si localizamos el nodo o una clave mayor.

Si hemos encontrado el nodo que queremos eliminar, será necesario modificar dos referencias:

- El campo *ant* del nodo que sigue al que se va a eliminar deberá apuntar al anterior al eliminado (*actual.sig.ant = actual.ant*)
- Además será necesario distinguir dos casos:
  - o si queremos borrar el primer nodo, y por tanto debemos modificar inicio, (*inicio = inicio.sig*)
  - o o bien si queremos borrar un nodo intermedio (*anterior.sig = actual.sig*)

La figura 3.21 ilustra el proceso (eliminación del nodo de clave 4).

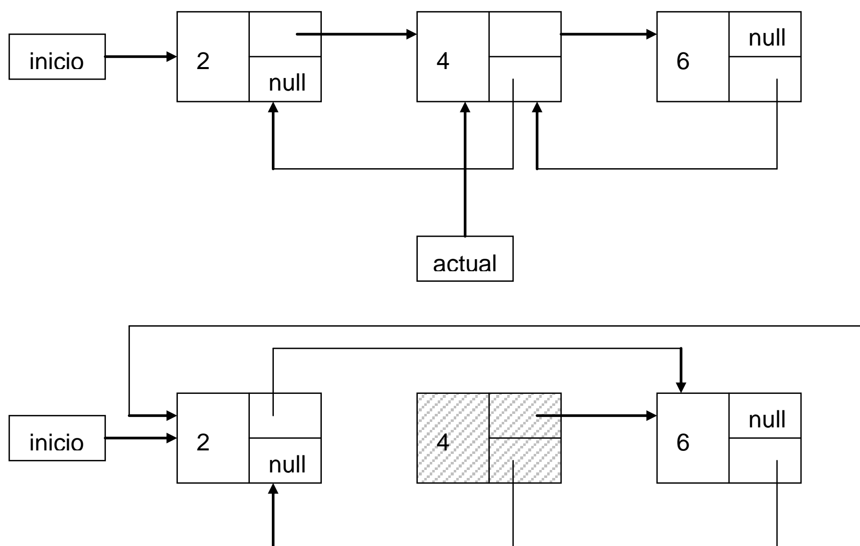

Figura 3.21. Eliminación de un nodo intermedio

La codificación es la siguiente:

```
public void eliminar (int clave) {
   NodoLista anterior, actual;
   boolean encontrado = false;
   anterior= inicio;
   actual= inicio;
   while ((actual != null) && !encontrado)
      if (actual.clave < clave) {
         anterior = actual;
         actual = actual.sig;
       }
      else encontrado = true;
   if (actual == null)
      System.out.println ("Error, el elemento no existe");
   else if (actual.clave > clave)
           System.out.println ("Error, el elemento no existe");
   else if (inicio == actual) {
         inicio = inicio.sig;
         inicio.ant = null;
      }
      else {
         anterior.sig = actual.sig;
         actual.sig.ant = anterior;
      }
}
```

3.7.3. Listas con cabecera ficticia y centinela.

Las técnicas denominadas *cabecera ficticia* y *centinela* se utilizan para mejorar la eficiencia en la codificación iterativa de las listas calificadas..

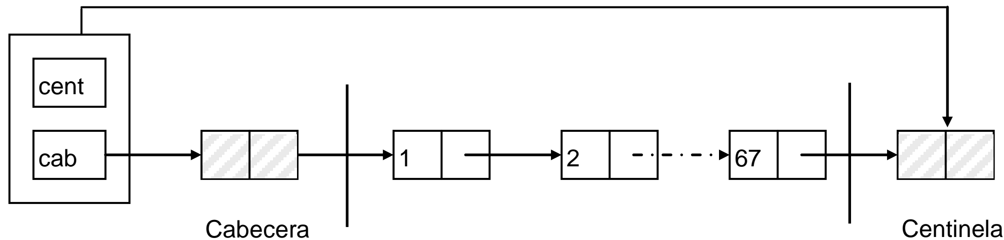

Figura 3.22. Utilización combinada de las técnicas de cabecera ficticia y centinela.

En esta técnica se utilizará la siguiente clase *NodoLista*:

```
class NodoLista {
   int clave;
   NodoLista sig;
   NodoLista (int dato) {
      clave = dato;
      sig = null;
   }
}
```

En cuanto a la clase Lista, se redefine como un registro de dos referencias a nodos de la lista: *cab* (cabecera), y *cent* (centinela).

```
public class Lista {
   NodoLista cab, cent;
   ...
}
```

La *cabecera ficticia* consiste en incorporar un nodo falso al principio de la lista. Cualquier proceso de búsqueda se iniciará con la referencia *anterior* apuntando al elemento ficticio y la referencia *actual* apuntando al primer elemento de la lista real (segundo nodo de la lista física). Con esto se elimina la excepción que se produce:

- Cuando se quiere insertar un elemento al principio de la lista.
- Cuando se va a eliminar el primer nodo de la lista.

La figura 3.23 explica esta técnica

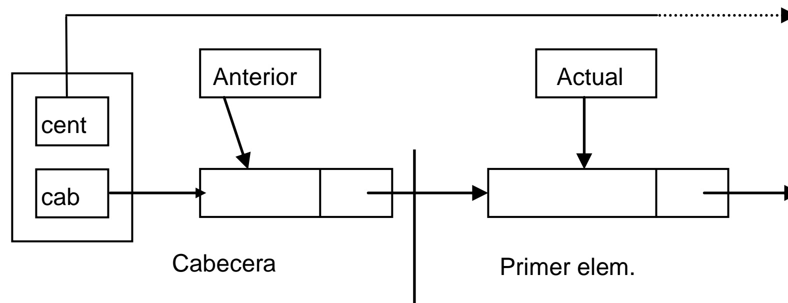

Figura 3.23. Técnica de cabecera ficticia con valores iniciales de las referencias anterior y siguiente

El campo *clave* del nodo apuntado por *cab* a veces se utiliza para guardar algún tipo de información relativa a la lista, por ejemplo el número de elementos de la misma.

La referencia centinela (*cent*) apunta a otro nodo ficticio (no contiene información de la lista) que se inserta al final. La técnica del centinela se basa, tanto en las operaciones de búsqueda y eliminación como en las de inserción, en copiar antes de la ejecución del tratamiento iterativo la clave pasada como argumento en el nodo centinela (*cent.clave = dato*). Con esto se consigue simplificar la lógica de las condiciones de finalización dado que siempre se va a encontrar la clave buscada. El cumplimiento de la condición (*actual == cent),* se interpreta en el sentido de que hemos llegado al final de la lista y la clave buscada no se encuentra realmente en la lista.

Combinando las dos técnicas anteriores se consigue una gran mejora de rendimiento en la localización de información de una lista calificada así como una simplificación de la lógica. El consumo adicional de espacio (necesario para alojar los nodos *cab* y *cent*) suele quedar compensado por las ventajas anteriormente expuestas.

3.7.3.1. Creación.

Para crear una lista con cabecera y centinela (constructor vacío), primero crearemos los dos nodos ficticios y después pondremos el centinela como siguiente de la cabecera:

```
Lista () {
   cab = new NodoLista (0);
   cent = new NodoLista (0);
   cab.sig = cent;
}
```

3.7.3.2.Recorrido completo.

Si lo que deseamos hacer es recorrer completamente una lista, como por ejemplo, para escribir todas las claves que contiene, tendremos que tener en cuenta que las claves contenidas en la cabecera y el centinela no forman parte de la lista real.

```
void imprimirLista () {
   NodoLista actual;

   actual = cab.sig;
   while (actual != cent) {
      System.out.print (actual.clave + " ");
      actual = actual.sig;
   }
   System.out.println (" FIN");
}
```

3.7.3.3. Búsqueda.

Para realizar una búsqueda de una clave en la lista con cabecera y centinela, utilizaremos las referencias auxiliares *anterior* (que apuntará inicialmente a la cabecera), y *actual* (que al principio apuntará al primer nodo de la lista, si ya hay nodos reales en la lista, y al centinela si está vacía), para recorrer la lista. Al finalizar el recorrido de la lista, devolveremos *true* si hemos encontrado la clave buscada, y *false* en caso de no haberla localizado (o bien hemos localizado una clave mayor, o bien hemos llegado al centinela).

```
   public boolean busqueda (int dato) {
      NodoLista anterior, actual;
      boolean resul = false;

      anterior = cab;
      actual = anterior.sig;
      cent.clave = dato;
      while (actual.clave < dato) {
         anterior = actual;
         actual = actual.sig;
      }
      if ((actual != cent) && (actual.clave == dato))
         resul = true;
      return resul;
   }
```

3.7.3.4. Inserción.

Para insertar un nuevo elemento en la lista, utilizaremos las referencias auxiliares *anterior* y *actual* para recorrer la lista, así como *aux* para crear el nuevo nodo. A continuación, copiamos la clave buscada en el centinela, y buscamos el hueco donde insertar el nuevo nodo. Una vez localizado el hueco (si la clave no existe), vamos a realizar la inserción siempre de la misma forma, ya que siempre insertaremos el nodo en la parte interior de la lista, nunca en un extremo de la misma.

```
public void insertar (int dato) {
   NodoLista anterior, actual, aux;
   anterior = cab;
   actual = anterior.sig;
   cent.clave = dato;
   while (actual.clave < dato) {
      anterior = actual;
      actual = actual.sig;
   }
   if ((actual.clave > dato) || (actual == cent)) {
      aux = new NodoLista (dato);
      aux.sig = actual;
      anterior.sig = aux;
   }
   else System.out.println ("Error, el elemento está repetido");
}
```

3.7.3.5. Eliminación.

De forma similar a lo que ocurría en la inserción, para eliminar un elemento de la lista, utilizaremos las referencias auxiliares *anterior* y *actual* para recorrer la lista. Una vez localizada la clave en la lista, verificaremos que no hemos llegado al centinela (en ese caso la clave no estaría en la lista), y procederemos a desenganchar el nodo correspondiente.

```
   public void eliminar (int dato) {
      NodoLista anterior, actual;
      anterior = cab;
      actual = anterior.sig;
      cent.clave = dato;
      while (actual.clave < dato) {
         anterior = actual;
         actual = actual.sig;
      }
      if ((actual == cent) || (actual.clave > dato))
         System.out.println ("Error, elemento inexistente");
      else anterior.sig = actual.sig;
   }
```

3.8. IMPLEMENTACIÓN DE LISTAS UTILIZANDO MATRICES

3.8.1. Listas densas.

A lo largo de los siguientes apartados veremos otra posible implementación de algunas de las listas analizadas hasta ahora. En particular, veremos como contruir un ejemplo de lista ordinal (una pila), así como una lista calificada ordenada.

3.8.1.1.Lista densa ordinal

Si deseamos construir una pila utilizando una matriz, podríamos utilizar la siguiente estructura:

Ejemplo de pila con 5 elementos (y como máximo 10 elementos):

0	1	2	3	4	5	6	7	8	9
1	7	6	2	7	0	0	0	0	0

numNodos = 5
N = 10

Como en las pilas los elementos se insertan y eliminan por el mismo extremo, consideraremos que el primer elemento se apilará en la posición 0 de la matriz, el segundo en la posición 1, y así sucesivamente.

Utilizaremos la variable miembro *numNodos*, para saber cuál es el último nodo apilado (*numNodos – 1,* que sería el primero a desapilar), y si hay espacio todavía para apilar nuevos elementos.

Para implementar el TadPila con la misma interfaz que hemos visto en el tema de Tipos Abstractos de Datos (y en la implementación mediante una lista enlazada), podríamos utilizar las siguientes variables miembros y constructor:

```
public class TadPila implements Pila {
   int [ ] matriz;
   final int N = 10;
   int numNodos;
   TadPila () {
      matriz = new int [N];
      numNodos = 0;
   }
   .....
}
```

Como ya se ha indicado, *numNodos* representaría el número de elementos que se han apilado hasta ahora. A su vez, la variable miembro *matriz* contendrá los elementos de la pila, y *N* el número máximo de elementos que puede llegar a contener.

A continuación, se desarrollan las operaciones *apilar*, *desapilar* y *pilaVacia*. Obsérvese que la operación *apilar* devolverá un mensaje de error si *numNodos == N* (la pila está llena), y *desapilar* si *numNodos == 0* (la pila está vacía).

Para comprobar si la pila está vacía es suficiente con verificar el valor de *numNodos*.

```
public boolean pilaVacia () {
   return numNodos == 0;
}
public void apilar (int dato) {
   if (numNodos < N) {
      matriz [numNodos] = dato;
      numNodos++;
   }
   else System.out.println ("la pila está llena");
}
public int desapilar () throws PilaVacia {
   int resul = -9999;
   if (numNodos != 0) {
      numNodos--;
      resul = matriz [numNodos];
   }
   else throw new PilaVacia ("la pila está vacía");
   return resul;
}
```

3.8.1.2. Lista densa calificada ordenada.

Para construir una lista calificada ordenada utilizando una matriz, podemos plantear las siguientes variables miembro y constructor (para una lista densa):

```
public class Lista {
   int [ ] matriz;
   final int N = 10;
   int numNodos;
   Lista () {
      matriz = new int [N];
      numNodos = 0;
   }
...
}
```

Un posible ejemplo de lista densa calificada en la que se han insertado cinco elementos sería:

0	1	2	3	4	5	6	7	8	9
1	3	4	7	9	0	0	0	0	0

numNodos = 5
N = 10

A continuación desarrollaremos los métodos de objeto de la clase *Lista*.

3.8.1.2.1. Búsqueda.

Por ejemplo, para realizar la búsqueda de una clave, recorreremos la *matriz* hasta encontrar una clave mayor o igual que la buscada, o bien llegar hasta el final de la lista. Se desarrollan dos métodos: *busqueda (int dato)*, que si la lista no está vacía, invoca a otro auxiliar recursivo privado (*busqueda (int i, int dato)* que busca el elemento por la *matriz*:

```
   public boolean busqueda (int dato) {
      boolean resul = false;
      if (numNodos != 0)
         resul = busqueda (0, dato);
      return resul;
   }

   private boolean busqueda (int i, int dato) {
      boolean resul = false;
      if (i < numNodos)
         if (matriz[i] < dato)
            resul = busqueda (i+1, dato);
         else if (matriz [i] == dato)
               resul = true;
      return resul;
   }
```

3.8.1.2.2. Inserción.

Para realizar la inserción en una lista densa calificada ordenada, primero se comprueba si hay espacio para el nuevo nodo con el método *insertar (int dato)* (comprobando si *numNodos* < *N*), y en caso afirmativo, se utiliza un método auxiliar estático privado (*insertar (int i, int dato)*), que recorre la lista hasta localizar la posición en la que se debe insertar un nuevo elemento.

```
public void insertar (int dato) {
   if (numNodos < N)
      insertar (0, dato);
   else System.out.println ("la lista está llena");
}
private void insertar (int i, int dato) {
   if (i == numNodos) {
      matriz [numNodos] = dato;
      numNodos++;
   }
   else if (matriz[i] < dato)
         insertar (i+1, dato);
      else if (matriz [i] > dato) {
            for (int j = numNodos; j > i; j--)
               matriz [j] = matriz [j-1];
            matriz [i] = dato;
            numNodos++;
         }
         else System.out.println ("la clave ya existe");
}
```

Por ejemplo, si tenemos originalmente la siguiente lista ordenada:

0	1	2	3	4	5	6	7	8	9
1	3	4	7	9	0	0	0	0	0

numNodos = 5

Si fuésemos a insertar un elemento mayor que el último, tan solo lo colocaríamos en la posición numNodos, e incrementaríamos el valor de dicha variable:

0	1	2	3	4	5	6	7	8	9
1	3	4	7	9	12	0	0	0	0

numNodos = 6

Sin embargo, si queremos insertar un elemento por la parte central de la lista (por ejemplo, el 5), desplazaremos todos los elementos a partir de la posición 3 una posición a la derecha e incrementaremos el valor de *numNodos*:

0	1	2	3	4	5	6	7	8	9
1	3	4	5	7	9	12	0	0	0

numNodos = 7

3.8.1.2.3. Eliminación.

Para realizar la eliminación en una lista densa calificada ordenada, primero se comprueba si hay algún elemento con el método *eliminar (int dato)* (comprobando si *numNodos > 0*), y en caso afirmativo, se utiliza un método auxiliar estático privado (*eliminar (int i, int dato)*), que recorre la lista hasta localizar la posición en la que se debe eliminar el elemento.

```
public void eliminar (int dato) {
   if (numNodos != 0)
      eliminar (0, dato);
   else System.out.println ("la lista está vacía");
}
private void eliminar (int i, int dato) {
   if (i < numNodos)
      if (matriz[i] < dato)
         eliminar (i+1, dato);
      else if (matriz [i] > dato)
            System.out.println ("la clave no existe");
         else {
            for (int j = i; j < numNodos-1; j++)
               matriz [j] = matriz [j+1];
            numNodos--;
         }
}
```

Si partimos del ejemplo anterior:

0	1	2	3	4	5	6	7	8	9
1	3	4	5	7	9	12	0	0	0

numNodos = 7

Si queremos eliminar el último elemento (12), tan solo sería necesario cambiar el valor de *numNodos*:

0	1	2	3	4	5	6	7	8	9
1	3	4	5	7	9	12	0	0	0

numNodos = 6

Sin embargo, si deseamos eliminar un elemento situado por la parte central de la lista (por ejemplo, el 4), desplazaríamos hacia la izquierda todos los elementos a partir de la posición 3 y decrementaríamos el valor de *numNodos*.

0	1	2	3	4	5	6	7	8	9
1	3	5	7	9	9	12	0	0	0

numNodos = 5

Obsérvese que en ninguno de los dos casos el último elemento en realidad no se elimina, pero como se ha modificado *numNodos* (ahora su valor es 5), las posiciones 5 y 6 no se tendrán en cuenta a menos que se inserte un nuevo elemento.

3.8.1.2.4. Ejemplos de recorrido completo.

Por ejemplo, utilizando la anterior estructura, podríamos realizar un recorrido completo (escribir el contenido de una lista densa) con el siguiente método estático:

```
static void escribirLista (Lista lista) {
    for (int i = 0; i < lista.numNodos; i++)
        System.out.print (lista.matriz [i] + "  ");
    System.out.println ("FIN");
}
```

En el siguiente ejemplo, recorremos completa la lista sumando los elementos que contiene:

```
static int sumaElementos (Lista lista) {
    int resul = 0;
    for (int i = 0; i <lista.numNodos; i++)
        resul = resul + lista.matriz [i] ;
    return resul;
}
```

3.8.2. Lista enlazada sobre matriz

Al principio del tema vimos el siguiente ejemplo basado en una matriz de registros.

	0	1	2	3	4	5	6	7	8
clave	1	10	77	12	26	21	11	13	18
sig	2	3	4	7	6	0	8	5	0

Figura 3.24. Ejemplo de implementación de lista enlazada mediante una estructura estática (matriz).

Su interpretación es la siguiente:

- Las claves de la lista se representan en el primero de los campos.
- El primer elemento del *vector* (índice 0) es especial. Su primer campo hace referencia al primer nodo de la lista (el que ocupa el primer campo de la posición 1, es decir 10) y el segundo la posición del primer "hueco" (índice 2 cuyo contenido –77– es irrelevante).
- El segundo campo de cada nodo indica la posición (índice) del nodo que le sigue, salvo que se trate de un cero que indicaría el final de la lista de nodos.
- El segundo campo de cada hueco indicaría la posición del siguiente hueco, salvo que se trate de un cero que indicaría el final de la lista de huecos.

Así pues, la estructura definida en la figura 3.24., permitiría almacenar hasta 8 elementos en la lista (índices 1 a 8). En la situación actual almacenaría la secuencia de datos de la figura 3.4: 10, 12, 13 y 21 (índices 1, 3, 7 y 5, respectivamente).

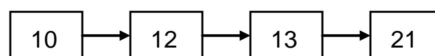

Figura 3.25. Lista enlazada correspondiente a la figura 3.24.

Y aún quedaría espacio para almacenar otros 4 datos (el primer espacio disponible es el nodo de índice 2 y el último el de índice 8).

Para poder utilizar una matriz para implementar una lista calificada ordenada, pordemos utilizar la estructura y constructor de la página siguiente.

```
class NodoLista {
   int clave, sig;
   NodoLista () {
      clave = 0;
      sig = 0;
   }
}
```

```
public class Lista {
   final int NULL = 0, N = 9;
   NodoLista [] matriz;
   Lista () {
      int i;
      matriz = new NodoLista [N];
      for (i = 0; i < N-1; i++) {
         matriz [i] = new NodoLista ();
         matriz [i].sig = i + 1;
      }
      matriz [i] = new NodoLista ();
   }
   ....
}
```

Utilizaremos la clase *NodoLista* para incluir los campos *clave* y *sig*. La clase *Lista* contendrá las constantes *N* (con valor 9, que representa el número máximo de nodos de la lista + 1) y *NULL* (con valor 0, que representa la posición vacía, equivalente a *null* en las listas enlazadas); además incluye la variable miembro *matriz* (que es un vector de elementos de la clase *NodoLista*), que contiene la lista vacía.

A continuación desarrollaremos los métodos de objeto de la clase *Lista*.

3.8.2.1.1. Búsqueda.

Para realizar la búsqueda de una clave, recorreremos la *matriz* siguiendo los enlaces hasta encontrar una clave mayor o igual que la buscada, o bien llegar hasta el final de la lista. Se desarrollan dos métodos: *busqueda (int dato)*, que si la lista no está vacía, invoca a otro auxiliar recursivo privado (*buscar (int pos, int dato)* pasándole como argumento la posición del primer elemento de la lista. El método buscar se encarga de localizar si el elemento aparece en la *matriz* o no:

```
   public boolean busqueda (int dato) {
      boolean resul = false;
      int pos = matriz [0].clave;
      if (pos != 0)
         resul = buscar (pos, dato);
      return resul;
   }
```

```
private boolean buscar (int pos, int dato) {
    boolean resul = false;
    if (matriz [pos].clave < dato) {
      if (matriz [pos].sig != 0)
        resul = buscar (matriz [pos].sig, dato);
    }
    else if (matriz [pos].clave == dato)
        resul = true;
    return resul;
  }
```

3.8.2.1.2. Inserción.

Para realizar la inserción en una lista calificada ordenada enlazada sobre matriz, primero se comprueba en el método *insertar (int dato)*:

- si la lista está llena (la lista de huecos estaría vacía: *matriz [0].sig == NULL*), en cuyo caso se produciría un mensaje de error,
- o bien si la lista está vacía vacía (comprobando si *matriz [0].clave != 0*), en dicho caso se inserta directamente el elemento utilizando el método auxiliar inser,
- en cualquier otro caso, se llama a un método auxiliar estático privado (*insertar (int pos, int ant, int dato)*), que recorre la lista hasta localizar la posición en la que se debe insertar un nuevo elemento (y volveríamos a llamar al método auxiliar *inser*).

```
  public void insertar (int dato) {
    if (matriz [0].sig != NULL) {
      int pos = matriz [0].clave;
      if (pos != 0)
        insertar (matriz [0].clave, 0, dato);
      else inser (0, 0, dato);
    }
    else System.out.println ("lista llena");
  }
  private void insertar (int pos, int ant, int dato) {
    if (matriz [pos].clave < dato)
      if (matriz [pos].sig != 0)
        insertar (matriz [pos].sig, pos, dato);
      else inser (0, pos, dato);
    else if (matriz [pos].clave > dato)
        inser (pos, ant, dato);
        else System.out.println ("la clave ya existe");
  }
```

```
private void inser (int pos, int ant, int dato) {
   int nuevo = matriz [0].sig;
   matriz [0].sig = matriz [nuevo].sig;
   matriz [nuevo].clave = dato;
   if (ant != 0) {
      matriz [nuevo].sig = pos;
      matriz [ant].sig = nuevo;
   }
   else {
      int sig = matriz [0].clave;
      matriz [0].clave = nuevo;
      if (pos == 0)
         matriz [nuevo].sig = 0;
      else matriz [nuevo].sig = sig;
   }
}
```

3.8.2.1.3. Eliminación.

Para realizar la eliminación de un elemento, se ha optado por hacer un método iterativo. Primero se verifica si la lista está vacía, y en caso contrario, se recorre la lista hasta encontrar la clave buscada (desenganchando el elemento de la lista de claves, y enlazándolo en la lista de huecos), o bien una clave mayor que la buscada (en cuyo caso se producirá un mensaje de error).

```
public void eliminar (int d) {
   int ant, pos, posAnt = 0;
   if (matriz [0].clave != NULL) {
      pos = matriz [0].clave;
      ant = matriz [pos].clave;
      while (ant < d) {
         posAnt = pos;
         pos = matriz [pos].sig;
         ant = matriz [pos].clave;
      }
      if (ant == d) {
         if (pos == matriz [0].clave)
            matriz [0].clave = matriz [pos].sig;
         else matriz [posAnt].sig = matriz[pos].sig;
         matriz [pos].sig = matriz [0].sig;
         matriz [0].sig = pos;
      }
      else System.out.println ("la clave no existe");
   }
   else System.out.println ("Error. La lista está vacía");
}
```

3.8.2.1.4. Ejemplos de recorrido completo.

Por ejemplo, utilizando la anterior estructura, podríamos realizar un recorrido completo (escribir el contenido de una lista enlazada sobre matriz) con el siguiente método estático:

```
static void escribirLista (Lista lista) {
   int  i = lista.matriz [0].clave;
   while (i != 0) {
      System.out.print (lista.matriz [i].clave+" ");
      i = lista.matriz [i].sig;
   }
   System.out.println (“FIN”);
}
```

En el siguiente ejemplo, recorremos completa la lista sumando los elementos que contiene:

```
static int sumaElementos (Lista lista) {
   int  i = lista.matriz [0].clave, resul = 0;
   while (i != 0) {
      resul = resul + lista.matriz [i].clave;
      i = lista.matriz [i].sig;
   }
   return resul;
}
```

3.9. UTILIZACIÓN DE UN TAD LISTA.

Los algoritmos explicados en las secciones anteriores implican el conocimiento interno de la estructura de la lista. Otra opción consiste en que el programador dispusiera de un Tipo Abstracto de Datos del que, dada su característica de **ocultamiento**, se desconoce absolutamente su implementación y, en consecuencia, solo es posible disponer de sus funcionalidades mediante un conjunto de especificaciones. Estas deberían ser, al menos, las siguientes:

```
public interface Lista {
   void crearNodo ();
      /*Crea un nuevo nodo en el tadLista*/
   int devolverClave ();
      /*Devuelve la clave contenida en el nodo del tadLista*/
   NodoLista devolverSiguiente ();
      /*Devuelve una referencia al siguiente del tadLista*/
   void asignarClave (int dato);
      /*Asigna el dato al primer nodo del TadLista*/
   void asignarReferencia (NodoLista referencia);
      /*Hace que el primer nodo del TadLista apunte al mismo sitio que
referencia*/
   void asignarReferenciaSiguiente (NodoLista referenciaNueva);
      /*Hace que el siguiente del nodo actual apunte ahora al mismo sitio que
referenciaNueva*/
   void asignarNulo ();
      /*Hace que el tadLista tome el valor null*/
   boolean esNulo ();
      /*Devuelve true si el inicio del TadLista tiene valor null; false en caso
contrario*/
   boolean esIgual (NodoLista referencia);
      /*Devuelve true si referencia apunta al mismo sitio que el tadLista, false
en caso contrario*/
}
```

A continuación se muestra un sencillo ejemplo que permite contar el número de nodos de una lista del TAD lista descrito arriba.

```
   static int contar (TadLista lista) {
      TadLista aux = new TadLista ();
      int resul;
      if (!lista.esNulo ()) {
         aux.asignarReferencia (lista.devolverSiguiente ());
         resul = 1 + contar (aux);
      }
      else resul = 0;
      return resul;
   }
```

3.9.1. Implementación del TAD Lista.

```
package tadLista;
public class TadLista implements Lista {
   NodoLista inicio;
   public TadLista () {
      inicio = null;
   }
   public void crearNodo () {
      /*Crea un nuevo nodo en el TadLista al principio de la lista*/
      inicio = new NodoLista (0, inicio);
   }
   public int devolverClave () {
      /*Devuelve la clave contenida en el nodo del tadLista*/
      return inicio.clave;
   }
   public NodoLista devolverSiguiente () {
      /*Devuelve una referencia al siguiente del TadLista*/
      return inicio.sig;
   }
   public NodoLista devolverReferencia () {
         /*Devuelve una referencia al primer nodo del TadLista*/
         return inicio;
      }
   public void asignarClave (int dato) {
      /*Asigna el dato al primer nodo del TadLista*/
      inicio.clave = dato;
   }
   public void asignarReferencia (NodoLista referencia) {
      /*Hace que el inicio del TadLista apunte al mismo sitio que referencia*/
      inicio = referencia;
   }
   public void asignarReferenciaSiguiente (NodoLista referenciaNueva) {
      /*Hace que el siguiente del nodo inicio apunte ahora al mismo sitio que
referenciaNueva*/
      inicio.sig = referenciaNueva;
   }
   public void asignarNulo () {
      /*Hace que el inicio del TadLista tome el valor null*/
      inicio = null;
   }
   public boolean esNulo () {
      /*Devuelve true si el inicio del TadLista tiene valor null; false en caso
contrario*/
      return inicio == null;
   }
   public boolean esIgual (NodoLista referencia) {
      /*Devuelve true si referencia apunta al mismo sitio que el inicio del
TadLista, false en caso contrario*/
      return inicio == referencia;
   }
}
```

TEMA 3 93

3.1. Conceptos Generales. 93

3.2. Implementación de listas. 95

3.3. Tratamiento de listas en java 97

3.4. Algoritmos básicos con Listas. 98

3.4.1. Recorrido completo. 98

3.5. Listas ordinales. 100

3.5.1. Pilas. 100

3.5.2. Colas. 102

3.6. Listas calificadas. 105

3.6.1. Listas calificadas no ordenadas. 105

3.6.2. Listas calificadas ordenadas. 112

3.7. Otras implementaciones. 118

3.7.1. Listas circulares (anillos). 118

3.7.2. Listas bidireccionales (doblemente enlazadas). 121

3.7.3. Listas con cabecera ficticia y centinela. 126

3.8. IMPLEMENTACIÓN DE LISTAS UTILIZANDO MATRICES 130

3.8.1. Listas densas. 130

3.8.2. Lista enlazada sobre matriz 136

3.9. Utilización de un TAD Lista. 141

3.9.1. Implementación del TAD Lista: 142