Estructuras de datos

¿Qué son las estructuras de datos?

Muchos algoritmos requieren una representación apropiada de los datos para lograr ser eficientes.

Algoritmos + Estructuras de Datos = Programas

Niklaus Wirth.

Una **estructura de datos** es una **representación de datos** junto con las **operaciones permitidas** sobre dichos datos.

Típicamente todas las estructuras de datos permiten **inserciones arbitrarias**. Las estructuras de datos varían en como permiten el **acceso a los miembros** del grupo. Algunas permiten tanto accesos como operaciones de **borrado** arbitrarios. Otras imponen restricciones, tales como permitir el acceso sólo al elemento más recientemente insertado, o al menos recientemente insertado.

Las estructuras de datos nos permiten lograr un importante objetivo de la programación orientada a objetos: **la reutilización de componentes**. Una vez que una estructura de datos ha sido implementada, puede ser utilizada una y otra vez en diversas aplicaciones.

El enfoque, **separación de la interfaz y la implementación**, es parte del paradigma de la programación orientada a objetos.

El usuario de la estructura de datos no necesita ver la implementación, sólo las operaciones disponibles. Esta es la parte de **ocultamiento y encapsulación**, de la programación orientada a objetos.

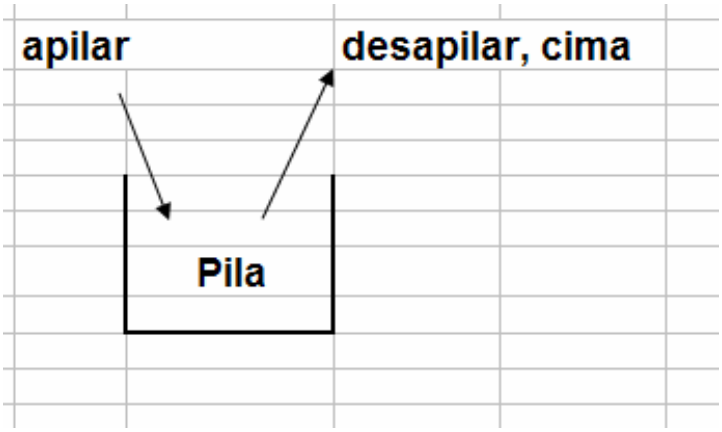
Otra parte importante de la programación orientada a objetos es **la abstracción**, Se deben pensar cuidadosamente el diseño de las estructuras de datos sin tener en cuenta la implementación. Esto hace la interfaz más limpia, más flexible, más reutilizable y generalmente más fácil de implementar.

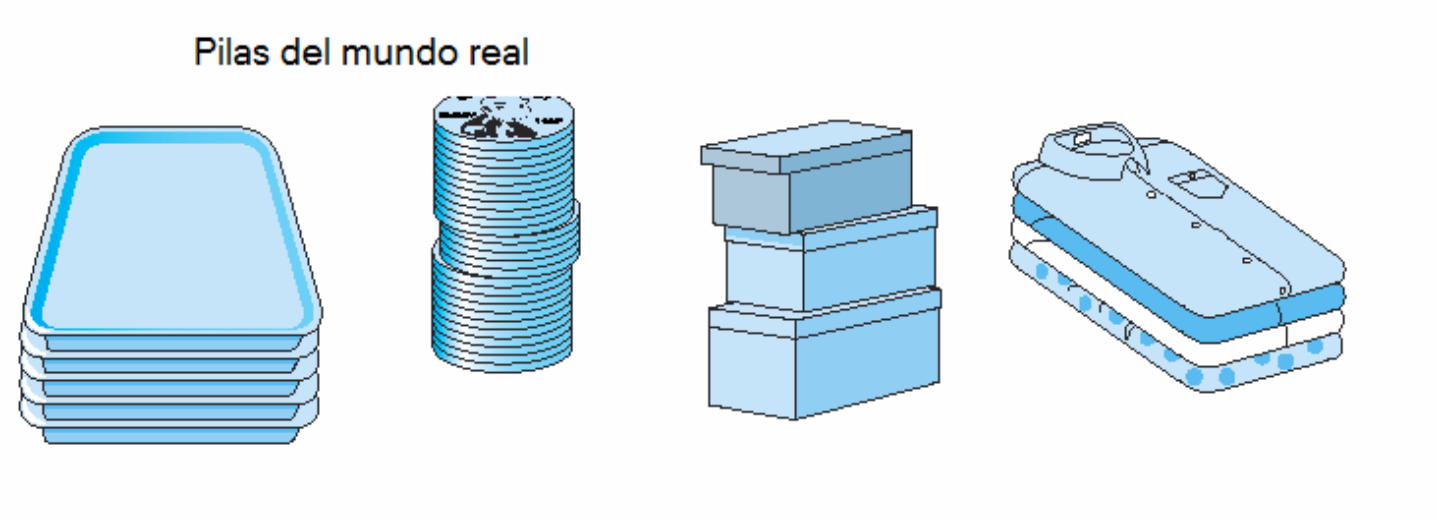
Pilas

Una pila es una estructura de datos **LIFO (Last In First Out)** en al cual el acceso está limitado al elemento más reciente insertado.

En una pila, las tres operaciones naturales de **insertar, eliminar y buscar**, se renombran por **apilar, desapilar y cima**.

Modelo de una pila





Lo que hace que las pilas sean útiles es que hay muchas aplicaciones en las que sólo es necesario acceder al elemento más reciente.

Las operaciones sobre las pilas deben ser de orden constante. **O(1).**

Interfaz para las pilas

```java
import com.practicas.excepciones.DesbordamientoInferior;

// Interfaz Pila
// ********************** OPERACIONES PUBLICAS **********************
// void apilar(x)          -> Inserta x
// void desapilar()        -> Elimina el ultimo elmento insertado
// Object cima()           -> Devuelve el ultimo elmento insertado
// Object cimaYDespilar()  -> Devuelve y elimina elemento mas reciente
// boolean esVacia()       -> Devuelve true si pila vacia, sino false
// void vaciar()           -> Elimina todos los elementos
// ******************************************************************
// ERRORES: cima, desapilar o cimaYDesapilar sobre pila vacia

public interface Pila {
     * Comprueba si la pila esta vacia
    boolean esVacia();

     * Obtiene el elemento mas recientemente insertado en pila No altera pila
    Object cima() throws DesbordamientoInferior;

     * Elimina el elemnto mas recientemente insertado

    void desapilar() throws DesbordamientoInferior;

     * Devuelve y elimina el elemento mas recientemente inserta en pila
    Object cimaYDesapilar() throws DesbordamientoInferior;

     * Inserta un elemento nuevo en la pila
    void apilar(Object x);

     * pone pila logicamente a vacio
    void vaciar();
}
```

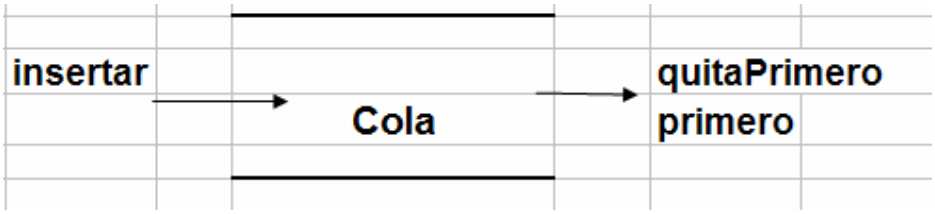
Colas

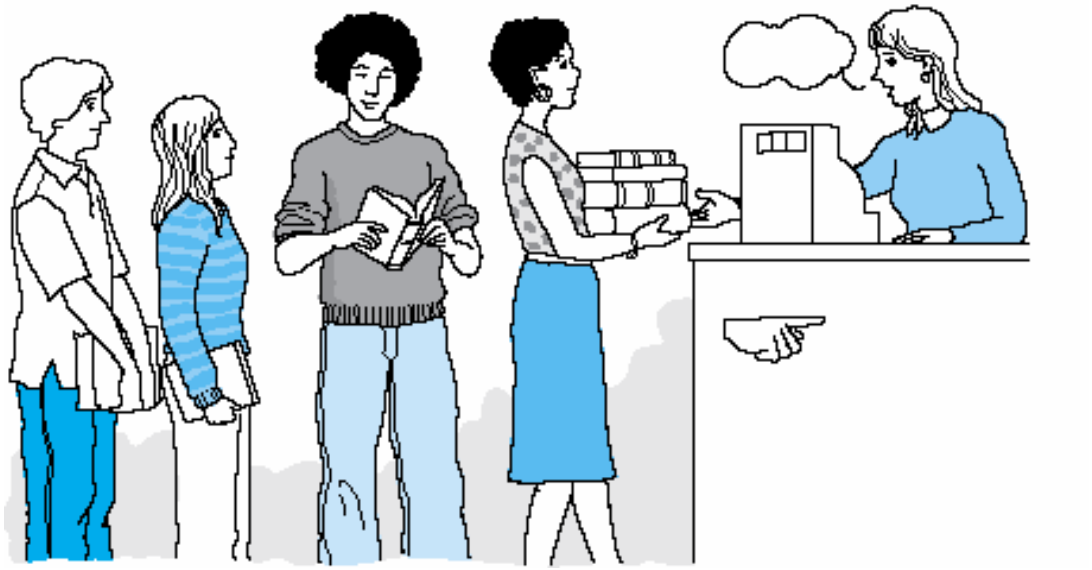
Una cola es una estructura de datos **FIFO (First In First Out)** en al cual el acceso está limitado al elemento menos recientemente insertado.

En una cola, las tres operaciones naturales de **insertar, eliminar y buscar**, se renombran por **insertar, quitarPrimero y primero**.

Las operaciones sobre las colas son de orden constate. **O(1).**

Modelo de una cola



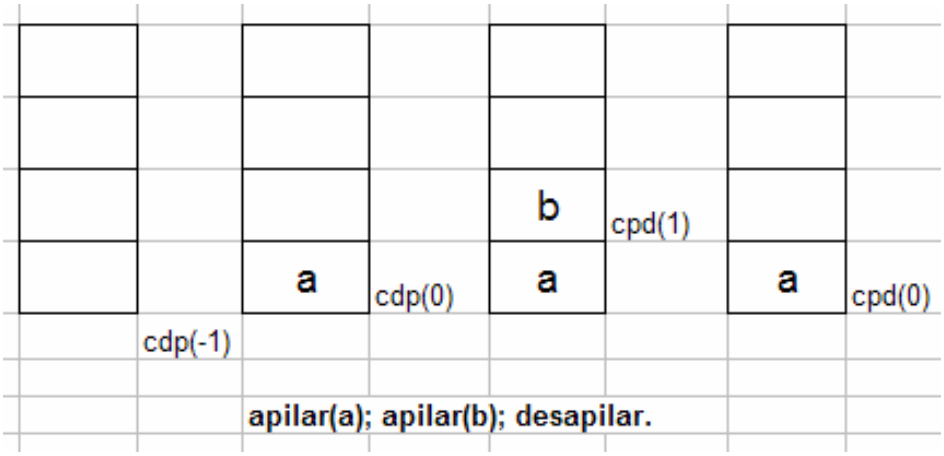


Interfaz para las colas

```
package com.practicas.estructurasdatos;

// Interfaz Cola
//
// CONSTRUCCION: sin ninguna inicialización
//
// ******************OPERACIONES PUBLICAS*************************************
// void insertar( x )       --> Inserta x
// Object primero( )        --> Devuelve el último elemento insertado
// Object quitarPrimero( )  --> Devuelve y elimina el último elemento insertado
// boolean esVacia( )       --> Devuelve true if empty; sino false
// void vaciar( )           --> Elimina todos los elementos
// ******************ERRORES**************************************************
// primero o quitarPrimero en una cola vacia
import com.practicas.excepciones.*;

public interface Cola {
     * Comprueba si la cola esta lógicamente vacia.
    boolean esVacia();

     * Devuelve el elemento insertado primero en la cola. No altera la cola.
    Object primero() throws DesbordamientoInferior;

     * Devuelve y elimina el elemento insertado primero en la cola. No altera la
    Object quitarPrimero() throws DesbordamientoInferior;

     * Inserta un elemento nuevo en la cola.
    void insertar(Object x);

     * Hace la cola vacia logicamente.
    void vaciar();
}
```

Pilas: Implementación dinámica basadas en un vector.

Una pila puede implementarse mediante un vector y un entero (cdp), que indica el indice del elemento situado en la cima de la pila




```
public class PilaVec implements Pila {
    // Atributos
    private Object vector[];

    private int cimaDePila;

    private static final int TAMAÑO = 10;

     * Creamos la pila cimaDePila=-1
    public PilaVec() {
        vector = new Object[TAMAÑO];
        cimaDePila = -1;
    }

     * Comprueba si pila vacia
    public boolean esVacia() {
        return cimaDePila == -1;
    }

     * Vaciar logicamente pila cimaDePila=-1;
    public void vaciar() {
        cimaDePila = -1;
    }
```

```
     * Devuelve el ultimo elemento insertado en pila
    public Object cima() throws DesbordamientoInferior {
        if (esVacia())
            throw new DesbordamientoInferior("pila vacia en metodo cima");
        else
            return vector[cimaDePila];
    }

     * Elimina el ultimo elmento insertado en la pila
    public void desapilar() throws DesbordamientoInferior {

     * Devuelve y elimina el ultimo elemento insertado.
    public Object cimaYDesapilar() throws DesbordamientoInferior {
        if (esVacia())
            throw new DesbordamientoInferior("cimaYDesapilar en pila vacia");
        else
            return vector[cimaDePila--];
    }

     * Inserta un nuevo elemento en la pila
    public void apilar(Object x) {
        if (cimaDePila + 1 == vector.length)
            duplicarVector();
        vector[++cimaDePila] = x;
    }

     * Duplicamos el vector cuando llegamos a TAMAÑO
    private void duplicarVector() {
        Object aux[] = new Object[vector.length * 2];
        for (int i = 0; i < vector.length; i++)
            aux[i] = vector[i];

        vector = aux;
    }

}
```

La técnica de duplicación del vector recibe el nombre de **Amortización.**

No afecta a la eficiencia en uso prolongado de la estructura. El coste de duplicar, que es de orden lineal, O(N), ya que una operación de duplicar un vector de N elementos esta precedida, al menos por N/2 operaciones de apilar que no necesitan la duplicación. Se puede repartir el coste O(N) de la duplicación entre las operaciones de apilar, aumentando su coste en sólo una constante.

```java
public final class PruebaPila {
    public static void main(String[] args) {
        Pila p = new PilaVec();
        for (int i = 0; i < 20; i++)
            p.apilar(new Integer(i));
        System.out.print("Contenido");
        try {
            for (;;)
                System.out.print(" " + p.cimaYDesapilar());
        } catch (DesbordamientoInferior e) {
        }

        System.out.println();
    }
}
```

==========

```
Contenido 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

Colas: Implementación dinámica basadas en un vector.

Con un solo entero: *fin*

La forma más sencilla de implementar una cola consiste en almacenar sus elementos en un vector, colocando el elemento en cabeza en la primera posición del mismo (indice 0). Si ***fin***, representa la posición del último elemento de la cola, entonces para insertar un elemento bastaría con incrementar ***fin***. Esta implantación presenta el problema de que ***quitarPimero*** seria muy costosa, forzando en esta operación a desplazar todos los elementos del vector una vez eliminado el primero.

Con dos enteros: *fin, cabeza*

El problema de la implementación anterior se resuelve con otro entro entero: ***cabeza***. Basta incrementarlo en la operación de ***quitarPrimero***.

Esta solución aún presenta un problema, después de varias ejecuciones de ***quitarPrimero***, no podríamos añadir más elementos, aunque la cola (el vector que la sustenta) no esté realmente llena.

Implementación de las colas con la implementación circular.

Se utilizan tres enteros: ***fin, cabeza, tamaño***

		fin					
esVacia()							
	tamaño=0	cabeza					
		fin					
insertar(a)		**a**					
	tamaño=1	cabeza					
			fin				
insertar(b)		**a**	**b**				
	tamaño=2	cabeza					
			fin				
quitarPrimero()			**b**				
	tamaño=1		cabeza				
			fin				
quitarPrimero()							
	tamaño=0		cabeza				
		.....	.....	.....	.....	.....	

					fin
Tras tres inserciones			**c**	**d**	**e**
tamaño=3			cabeza		
	fin				
insertar(f)	**f**		**c**	**d**	**e**
tamaño=4			cabeza		
	fin				
quitarPrimero()	**f**			**d**	**e**
tamaño=3				cabeza	
	fin				
quitarPrimero()	**f**				**e**
tamaño=2					cabeza
	fin				
quitarPrimero()	**f**				
tamaño=1	cabeza				

- La circularidad devuelve ***cabeza*** y ***fin*** al principio de vector cuando rebasan la última posición.

- Si la cola está llena, es necesario realizar la duplicación del vector.

```java
public class ColaVec implements Cola {

    public Object[] vector;

    public int tamañoCola;

    public int cabeza, fin;

    static final int TAMAÑO_POR_DEFECTO = 5;

     * Constructor cola
    public ColaVec() {
        vector = new Object[TAMAÑO_POR_DEFECTO];
        vaciar();
    }

     * Comprueba si la cola esta logicamente vacia.
    public boolean esVacia() {
        return tamañoCola == 0;
    }

     * Dejar la cola vacia logicamente
    public void vaciar() {
        tamañoCola = 0;
        cabeza = 0;
        fin = -1;
    }
     * Devuelve el elemento insertado primero en la cola. No altera la cola.
    public Object primero() throws DesbordamientoInferior {
        if (esVacia())
            throw new DesbordamientoInferior("Primero");
        return vector[cabeza];
    }

     * Desvuelve y elimina el primer elemento insertado en la cola Altera la
    public Object quitarPrimero() throws DesbordamientoInferior {
        if (esVacia())
            throw new DesbordamientoInferior("Quitar Primero");
        tamañoCola--;

        Object valor = vector[cabeza];
        cabeza = incrementar(cabeza);
        return valor;
    }
```

```
     * Inserta un elemento nuevo en la cola.
    public void insertar(Object x) {
        if (tamañoCola == vector.length)
            duplicarCola();
        fin = incrementar(fin);
        vector[fin] = x;
        tamañoCola++;
    }

    /**
     * Metodo privado para duplicar vector cuando cola llena
     */
    private void duplicarCola() {
        Object[] auxVector;

        auxVector = new Object[vector.length * 2];
        for (int i = 0; i < tamañoCola; i++, cabeza = incrementar(cabeza))
            auxVector[i] = vector[cabeza];

        vector = auxVector;
        cabeza = 0;
        fin = tamañoCola - 1;
    }

    /**
     * Metodo privado para incremento con circularidad
     */
    private int incrementar(int x) {
        if (++x == vector.length)
            x = 0;
        return x;
    }
}
```

==========

```
public final class PruebaCola {
    public static void main(String[] args) {
        Cola c = new ColaVec();
        for (int i = 0; i < 20; i++)
            c.insertar(new Integer(i));
        System.out.print("Contenido");
        try {
            for (;;)
                System.out.print(" " + c.quitarPrimero());
        } catch (DesbordamientoInferior e) {
        }

        System.out.println();
    }
}
```

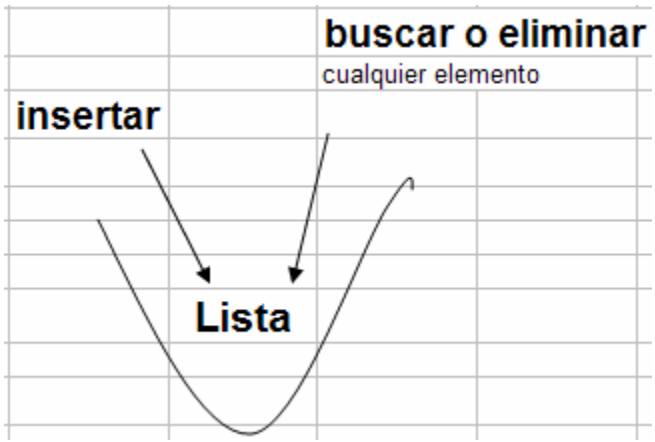
==========

```
Contenido 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

Listas enlazadas

En una lista enlazada, los elementos se almacenan de forma no contigua. En vez de en un vector de posiciones de memoria consecutiva.

Modelo de lista enlazada: Las entradas son arbitrarias. Cualquier elemento puede salir

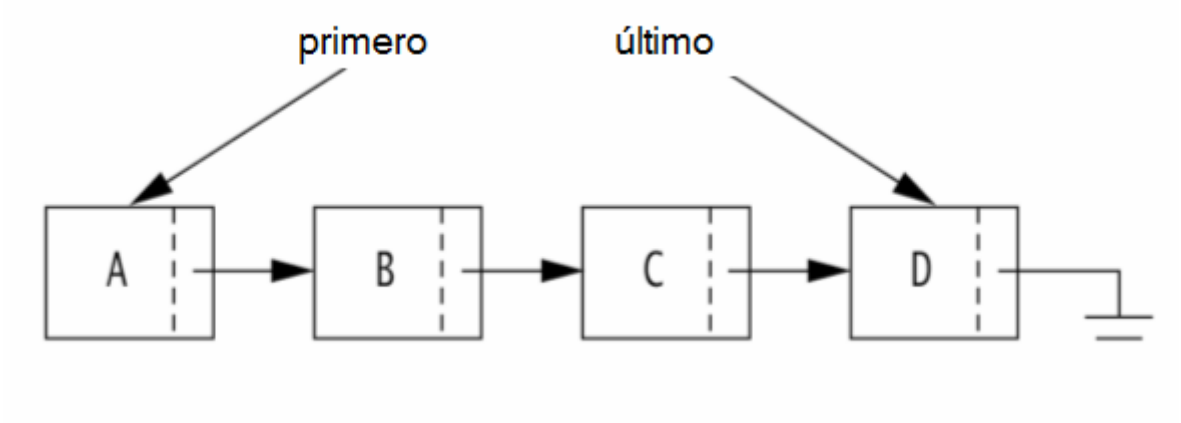


Cada elemento se almacena en **un nodo**, que contiene **un objeto** y **una referencia** al siguiente nodo de la lista.

Las listas enlazadas se utilizan para evitar movimientos de grandes cantidades de datos.

Almacenan elementos con **el coste añadido** de una referencia adicional por elemento.

```
class NodoLista {
    // Atributos
    Object dato;
    NodoLista siguiente;

    public NodoLista(Object elElemento) {
        this(elElemento, null);
    }
    public NodoLista(Object elElemento, NodoLista n) {
        dato = elElemento;
        siguiente = n;
    }
}
```



Interfaz para una lista abstracta

```
package com.practicas.estructurasdatos;

// Interfaz Lista
//
// Acceso mediante la clase ListaItr
//
// ******************OPERACIONES PÚBLICAS************************
// boolean esVacia( ) --> Return true si vacia; otro caso false
// void vaciar( )     --> Elimina todos los nodos
// ********************ERRORES***********************************
// No se tratan errores
/**
 * Protocolo para listas. Acceso a la lista se realiza mediante
 *  un objeto ListaItr.
 *
 * @see ListaItr
 */
public interface Lista {
     * Comprueba si la lista esta lógicamente vacia.
    boolean esVacia();

     * Vacia la lista lógicamente.
    void vaciar();
}
```

El acceso a la lista se realiza a través de una clase iteradora.

La clase lista tiene las operaciones que dan el estado de la lista.

El resto de las operaciones está en la clase iteradora.

Interfaz para una lista abstracta

```java
package com.practicas.estructurasdatos;

import com.practicas.excepciones.ElementoNoEncontrado;

// Interfaz ListaIter; mantiene "posicion actual"
//
// ***********************OPERACIONES PÚBLICAS*****************************
// void insertar( x )  --> Inserta x después de la posición Actual
// void eliminar( x )  --> Elimina x
// boolean buscar( x ) --> Posiciona la posición Actual para acceder a x
// void cero( )        --> Posiciona Actual al nodo cabecera
// void primero( )     --> Posiciona Actual al primero
// void avanzar( )     --> Avanza
// boolean estaDentro( ) --> True si es una posición valida en la lista
// Object recuperar( )   --> Devuelve el dato de la posición actual
// ***********************ERRORES*****************************************
// Excepciones generadas por accesos, inserciones o eliminaciones ilegales.
/**
 * Protocolo para los iteradores de listas utilizando un nodo cabecera.
 *
 * @see List
 */
public interface ListaIter {
     * Inserta después de la posición Actual. Actual se apunta a nodo actual.
    void insertar(Object x) throws ElementoNoEncontrado;

     * Establece la posición Actual en el primer nodo que contiene un item.
    boolean buscar(Object x);

     * Elimina la primera aparición de un valor. Actual se apunta al primer nodo
    void eliminar(Object x) throws ElementoNoEncontrado;

     * Comprueba que Acutal referencia un nodo de la lista.
    boolean estaDentro();

     * Devuelve el valor almacenado en la posición Actual.
    Object recuperar();

     * Posiciona Actual en al nodo cabecera.
    void cero();

     * Posiciona Actual en el primer nodo de la lista. Operación válida para
    void primero();

     * Avanza la posición Actual al siguiente nodo en la lista. Si la posición
    void avanzar();
}
```

Para añadir un nuevo elemento al final:

```
ultimo.siguiente = new NodoLista( ); //nuevo nodo lista
ultimo = ultimo.siguiente;           //ajusta el ultimo
ultimo.dato = x ;                    // coloca x en el nodo
ultimo.siguiente = null;             // Es el último, ajusta el siguiente
```

Iterar a través de una lista enlazada (básico):

```
for ( NodoLista p=analista.primero ; p ! = null ; p.siguiente)
    System.out.println( p.dato )
```

Iterar a través de una lista enlazada (con un iterador):

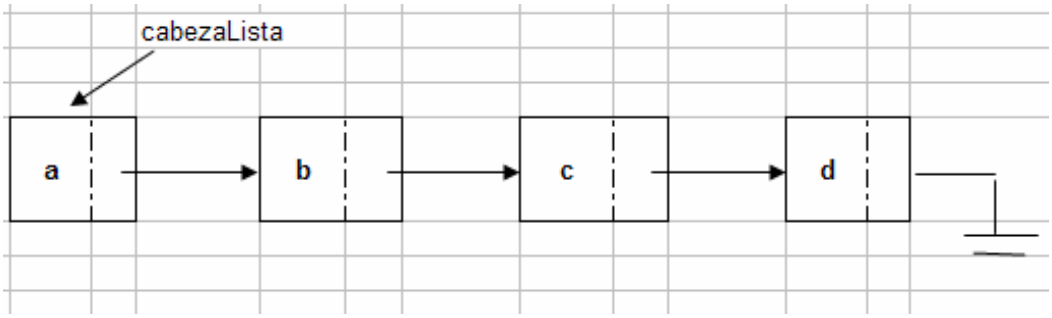
```
//recorrer la lista, utilizando la abstracción y el iterador…
ListaIter iter = new ListaIter (laLista); // cuidado…
for ( iter.primero( ) ; iter.estaDentro( ); iter.avanzar ( ) )
    System.out.println( iter.recuperar( ) )
```

```java
public final class PruebaLista {
    public static void main(String[] args) throws Exception {
        Lista laLista = new ListaEnlazada();
        ListaIter iter = new ListaEnlazadaIter(laLista);

        //Insertar repetidamente elementos por el principio
        for (int i = 0; i < 10; i++) {
            try {
                iter.insertar(new Integer(i));
            } catch (ElementoNoEncontrado e) {
            } // No puede ocurrir
            iter.cero();
        }
        System.out.print("Contenido");
        for (iter.primero(); iter.estaDentro(); iter.avanzar())
            System.out.print(" " + iter.recuperar());
        System.out.println(" fin");
    }
}
```

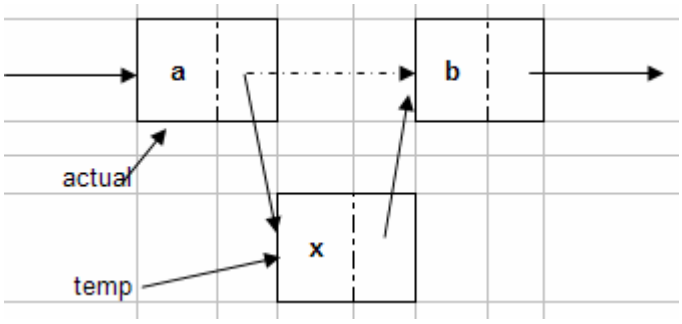
```
Contenido 9 8 7 6 5 4 3 2 1 0 fin
```

Implementación de las listas enlazadas

Lista enlazada básica



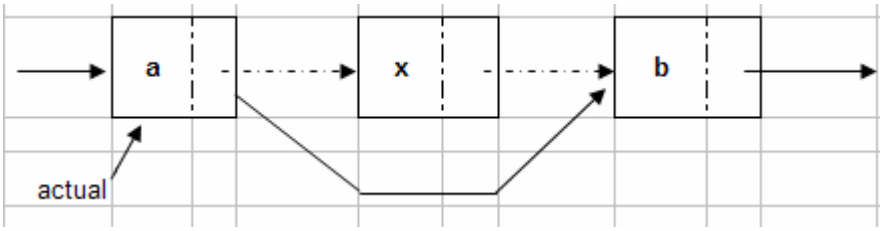
Inserción en una lista enlazada



tmp = new NodaLista(x, actual.siguiente); // creación de un nuevo nodo

actual.siguiente = tmp // el siguiente al nodo actual es x

Eliminación en una lista enlazada



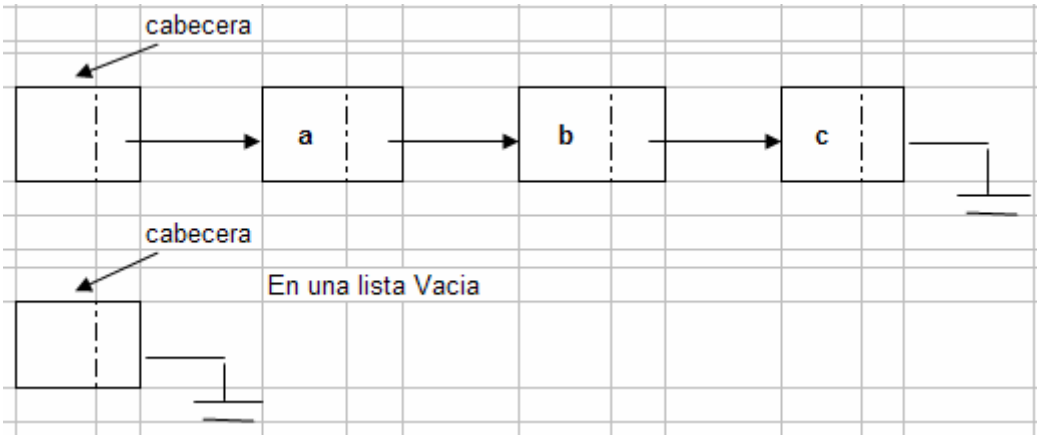
actual.siguiente = actual.siguiente.siguiente;

Las operaciones de inserción y eliminación son de orden constante O(1).

Nodos Cabecera

El nodo cabecera no guarda ningún dato, pero sirve para satisfacer el requerimiento de que cada nodo tenga uno anterior.

Los nodos cabecera nos evitan tener que tratar de forma explicita casos especiales, como la inserción de un elemento en la primera posición o la eliminación del primer elemento.



La clase ListaEnlazada

```
public class ListaEnlazada implements Lista {

    //  Atributos amistosos, luego ListaEnlazadaIter puede acceder
    NodoLista cabecera; // Referencia al nodo cabecera
    public ListaEnlazada() {
        cabecera = new NodoLista(null);
    }

    public boolean esVacia() {
        return cabecera.siguiente == null;
    }

    public void vaciar() {
        cabecera.siguiente = null;
    }
}
```

La clase ListaEnlazadaIter

```
public class ListaEnlazadaIter implements ListaIter {


    protected ListaEnlazada laLista; // Lista

    protected NodoLista actual; // Posición actual

    /**
     * Construye la lista Como resultado, la posición actual es el primer
     * elemento, a menos que la lista sea vacía, en cuyo caso la posición actual
     * es el elemento 0-ésimo.
     *
     * @param unaLista
     *            un objeto ListaEnlazada a la que este iterador está
     *            permanentemente ligado.
     */
    public ListaEnlazadaIter(ListaEnlazada unaLista) {
        laLista = unaLista;
        actual = laLista.esVacia() ? unaLista.cabecera
                : unaLista.cabecera.siguiente;
    }

    /**
     * Este constructor se proporciona por conveniencia. Si unaLista no es un
     * objeto ListaEnlazada, se lanzará una exceptción ClassCastException.
     * Enumerationcaso contrario, tiene el mismo comportamiento que el
     * constructor de arriba.
     */
    public ListaEnlazadaIter(Lista unaLista) throws Exception {
        this((ListaEnlazada) unaLista);
    }

    /**
     * Coloca la posición actual en el nodo cabecera.
     */
    public void cero() {
        actual = laLista.cabecera;
    }
```

```java
/**
 * Comprueba si la posición actual apunta a un elemento no válido.
 *
 * @return true si la posición actual no es null y no está apuntando al nodo
 *         cabecera
 */
public boolean estaDentro() {
    return actual != null && actual != laLista.cabecera;
}

/**
 * Avanzar la posición actual al nodo siguiente de la lista. Si la posición
 * actual es null, no hace nada. Enumerationesta rutina no se lanzan
 * excepciones porque su uso más habitual (dentro de un bucle for)
 * requeriría que el programador añadiera un bloque try/cach.
 */
public void avanzar() {
    if (actual != null)
        actual = actual.siguiente;
}

/**
 * Coloca la posición actual en el primer nodo. Esta operación es válida
 * para listas vacías.
 */
public void primero() {
    actual = laLista.cabecera.siguiente;
}

/**
 * Inserta tras la posición actual. actual apunta al nodo insertado con
 * éxito.
 *
 * @param x
 *            el elemento a insertar.
 * @exception ElementoNoEncontrado
 *                si la posición actual es null.
 */
public void insertar(Object x) throws ElementoNoEncontrado {
    if (actual == null)
        throw new ElementoNoEncontrado("Error de inserción");
    NodoLista nuevoNodo = new NodoLista(x, actual.siguiente);
    actual = actual.siguiente = nuevoNodo;
}
```

```java
    /**
     * Coloca la posición actual en el primer nodo que contiene un elemento. Si
     * no encontramos x, actual no se ve modificado.
     *
     * @param x
     *            el elemento a buscar.
     * @return true si encontramos el elemento x; si no false.
     */
    public boolean buscar(Object x) {
        NodoLista itr = laLista.cabecera.siguiente;
        while (itr != null && !itr.dato.equals(x))
            itr = itr.siguiente;
        if (itr == null)
            return false;
        actual = itr;
        return true;
    }
    /**
     * Elimina la primera aparición de un elemento. Si aparece en la lista se
     * coloca actual al principio; pero si no aparece en la lista, no se
     * modifica.
     *
     * @param x
     *            elemento a eliminar.
     * @exception ElementoNoEncontrado
     *                si no se encuentra.
     */
    public void eliminar(Object x) throws ElementoNoEncontrado {
        NodoLista itr = laLista.cabecera;
        while (itr.siguiente != null && !itr.siguiente.dato.equals(x))
            itr = itr.siguiente;
        if (itr.siguiente == null)
            throw new ElementoNoEncontrado("Fallo en eliminacion");

        itr.siguiente = itr.siguiente.siguiente;// Eliminación
        actual = laLista.cabecera;// Inicializa a null
    }
    /**
     * Devuelve el elemento almacenado en la posición actual.
     *
     * @return el elemento almacenado o null si la posición actual no está en la
     *         lista.
     */
    public Object recuperar() {
        return estaDentro() ? actual.dato : null;
    }
}
```

Listas enlazadas ordenadas

A partir de de ListaEnlazadaIter, podemos derivar la clase ListaEnlazadaIterOrd para mantener ordenados los elementos de una lista.

La diferencia fundamental entre la lista ordenada y la no ordenada es **la rutina de inserción**.

```
import com.practicas.excepciones.ElementoNoEncontrado;
import com.practicas.utilidades.Comparable;
import com.practicas.utilidades.Entero;

public class ListaEnlazadaIterOrd extends ListaEnlazadaIter {

    public ListaEnlazadaIterOrd(Lista unaLista) throws Exception {
        super(unaLista);
    }

    public void insertar(Object x) throws ElementoNoEncontrado {
        if (x instanceof Comparable) // Comparable de utilidades
            insertar((Comparable) x);
        else
            throw new ElementoNoEncontrado("ListaIterEnlazadaOrd "
                    + "insertar requiere un objeto de tipo Comparable");
    }

    // Inserta de forma ordenada
    public void insertar(Comparable x) {
        ListaEnlazadaIter ant = new ListaEnlazadaIter(laLista);
        ListaEnlazadaIter act = new ListaEnlazadaIter(laLista);

        ant.cero();
        act.primero();
        while (act.estaDentro() && ((Comparable) (act.recuperar())).menorQue(x)) {
            act.avanzar();
            ant.avanzar();
        }

        try {
            ant.insertar(x);
        } catch (Exception e) {
        }// no puede ocurrir
        actual = ant.actual;
    }
```

```
    public static void main(String args[]) throws Exception {
        Entero m = new Entero(5);
        Object obj = new Integer(5);
        ListaEnlazadaIterOrd itr = new ListaEnlazadaIterOrd(new ListaEnlazada())
        itr.insertar(m);
        try {
            itr.insertar(obj);
        } catch (Exception e) {
            System.out.println("Elemento1 no Comparable.");
        }

        obj = m;
        try {
            itr.insertar(obj);
        } catch (Exception e) {
            System.out.println("Elemento2 no Comparable.");
        }
    }
}
```

==========

```
Elemento1 no Comparable.
```

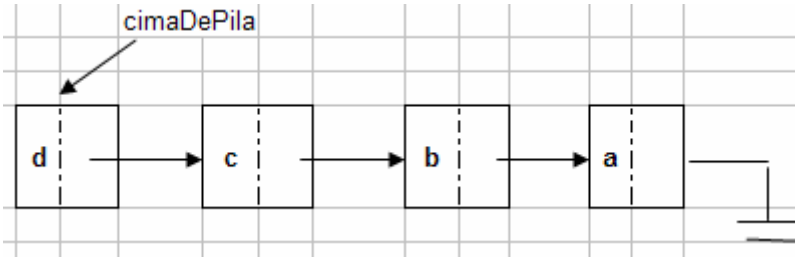
==========

```
public final class PruebaListaEnlazadaOrd {
    public static void main(String[] args) throws Exception {
        Lista laLista = new ListaEnlazada();
        ListaIter iter = new ListaEnlazadaIterOrd(laLista);

        //Insertar repetidamente elementos  aleatorios
        NumerosAleatorios r = new NumerosAleatorios();
        for (int i = 0; i < 15; i++) {
            try {
                iter.insertar(new Entero(r.randomInt(0, 20)));
            } catch (ElementoNoEncontrado e) {
            } // No puede ocurrir
            iter.cero();
        }
        System.out.print("Contenido");
        for (iter.primero(); iter.estaDentro(); iter.avanzar())
            System.out.print(" " + iter.recuperar());
        System.out.println(" fin");
    }
}
```

==========

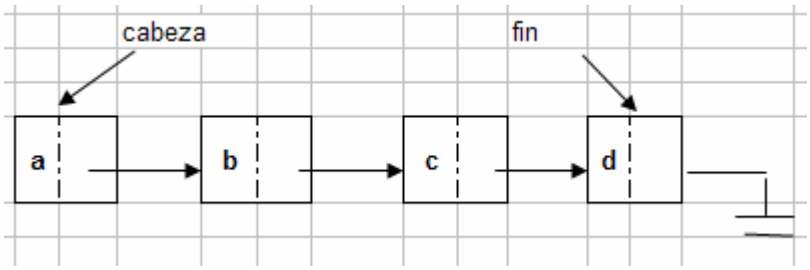
```
Contenido 0 0 1 3 3 7 7 8 8 10 10 10 13 13 14 14 18 18 18 19 fin
```

Pilas. Implementación con listas enlazadas




```
public class PilaLista implements Pila {

    // Atributos
    private NodoLista cimaDePila;

    public PilaLista() {
        cimaDePila = null;
    }
     * Comprueba si la pila esta vacia
    public boolean esVacia() {
        return cimaDePila == null;
    }
    /**
     * pone pila logicamente a vacio
     */
    public void vaciar() {
        cimaDePila = null;
    }
     * Obtiene el elemento mas recientemente insertado en pila.
    public Object cima() throws DesbordamientoInferior {
        if (esVacia())
            throw new DesbordamientoInferior("cima");
        return cimaDePila.dato;
    }
     * Inserta un elemento nuevo en la pila
    public void apilar(Object x) {
        cimaDePila = new NodoLista(x, cimaDePila);
    }
     * Elimina el elemnto mas recientemente insertado
    public void desapilar() throws DesbordamientoInferior {
        if (esVacia())
            throw new DesbordamientoInferior("desapilar");
        cimaDePila = cimaDePila.siguiente;
    }
     * Devuelve y elimina el elemento mas recientemente inserta en pila
    public Object cimaYDesapilar() throws DesbordamientoInferior {
        if (esVacia())
            throw new DesbordamientoInferior("cimaYDesapilar");
        Object datoCima = cimaDePila.dato;
        cimaDePila = cimaDePila.siguiente;
        return datoCima;
    }
}
```

Colas. Implementación con listas enlazadas



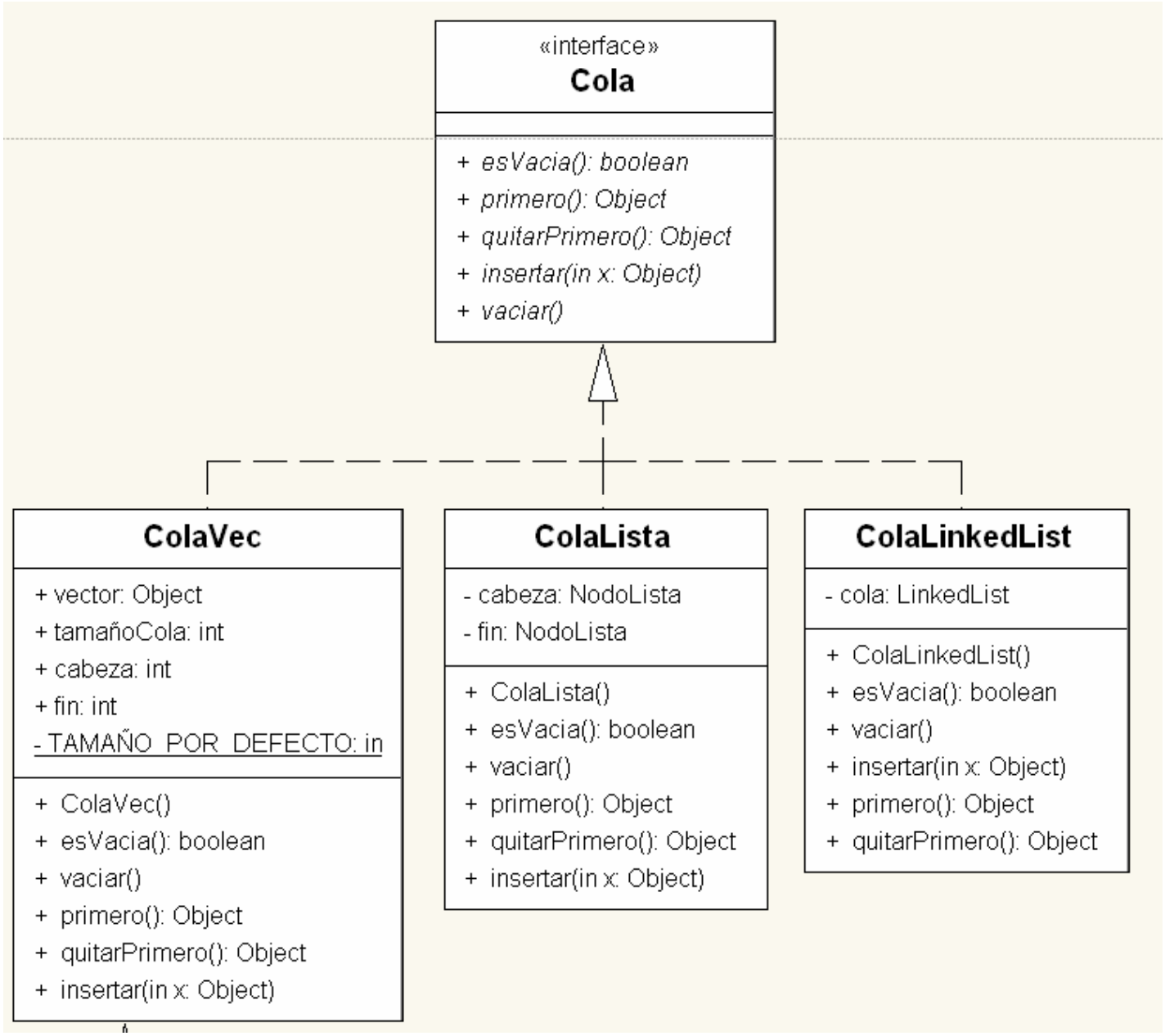

```
public class ColaLista implements Cola {

    private NodoLista cabeza;
    private NodoLista fin;

    /** Creación la cola */
    public ColaLista() {
        vaciar();
    }
     * Comprueba si la cola esta logicamente vacia
    public boolean esVacia() {
        return cabeza == null;
    }
     * Hace la cola vacia logicamente
    public void vaciar() {
        cabeza = fin = null;
    }
     * Devuelve el elemento insertado primero en la cola sin alterarla
    public Object primero() throws DesbordamientoInferior {
        if (esVacia())
            throw new DesbordamientoInferior("primero");
        return cabeza.dato;
    }
     * Desvuelve y elimina el primero de cola altera la cola
    public Object quitarPrimero() throws DesbordamientoInferior {
        if (esVacia())
            throw new DesbordamientoInferior("quitarPrimero");
        Object valorDevuelto = cabeza.dato;
        cabeza = cabeza.siguiente;
        return valorDevuelto;
    }
     * Inserta un elmento nuevo en la cola.
    public void insertar(Object x) {
        if (esVacia())
            fin = cabeza = new NodoLista(x);
        else
            fin = fin.siguiente = new NodoLista(x);
    }
}
```

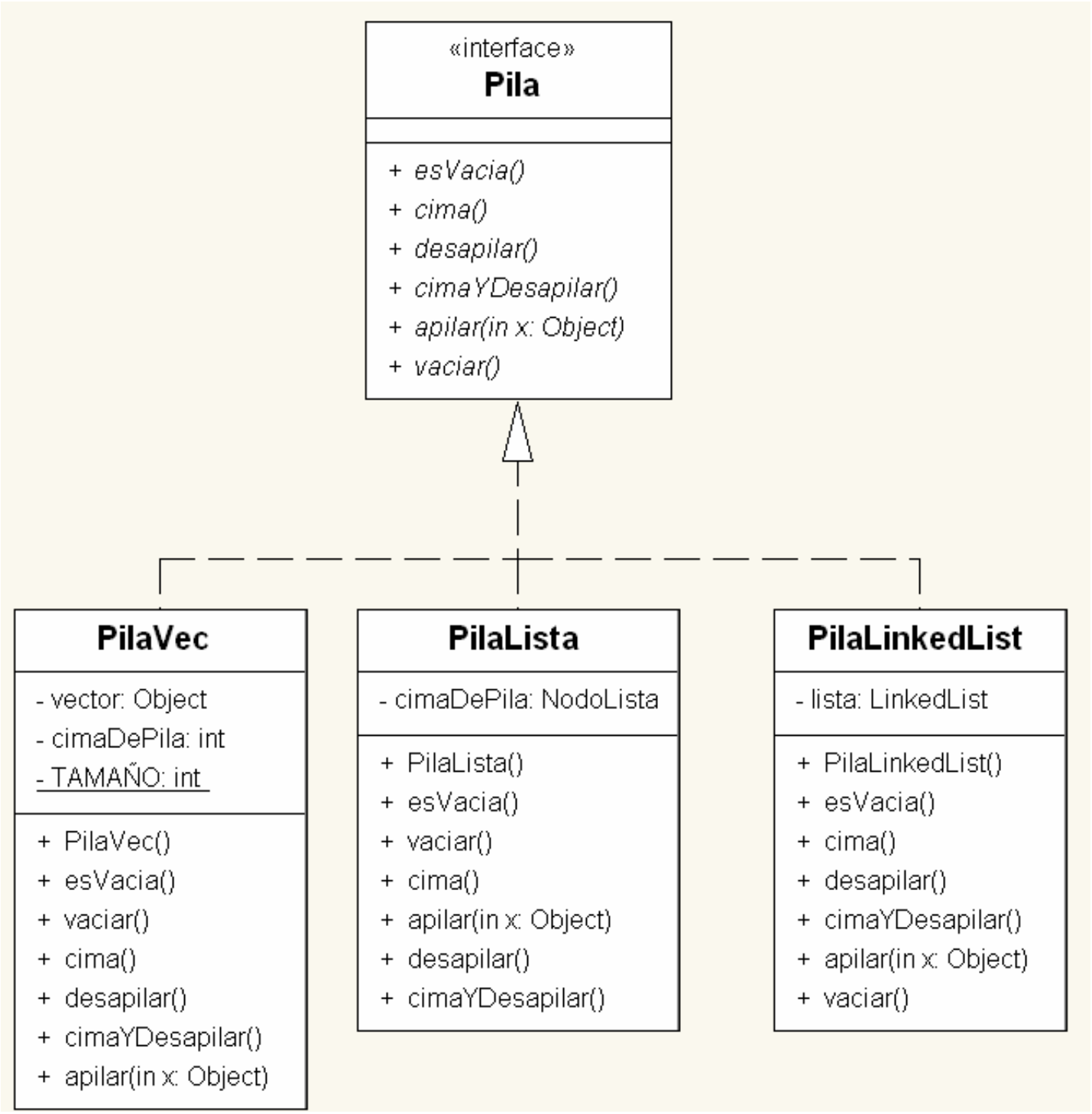
Pilas. Implementación con LinkedList

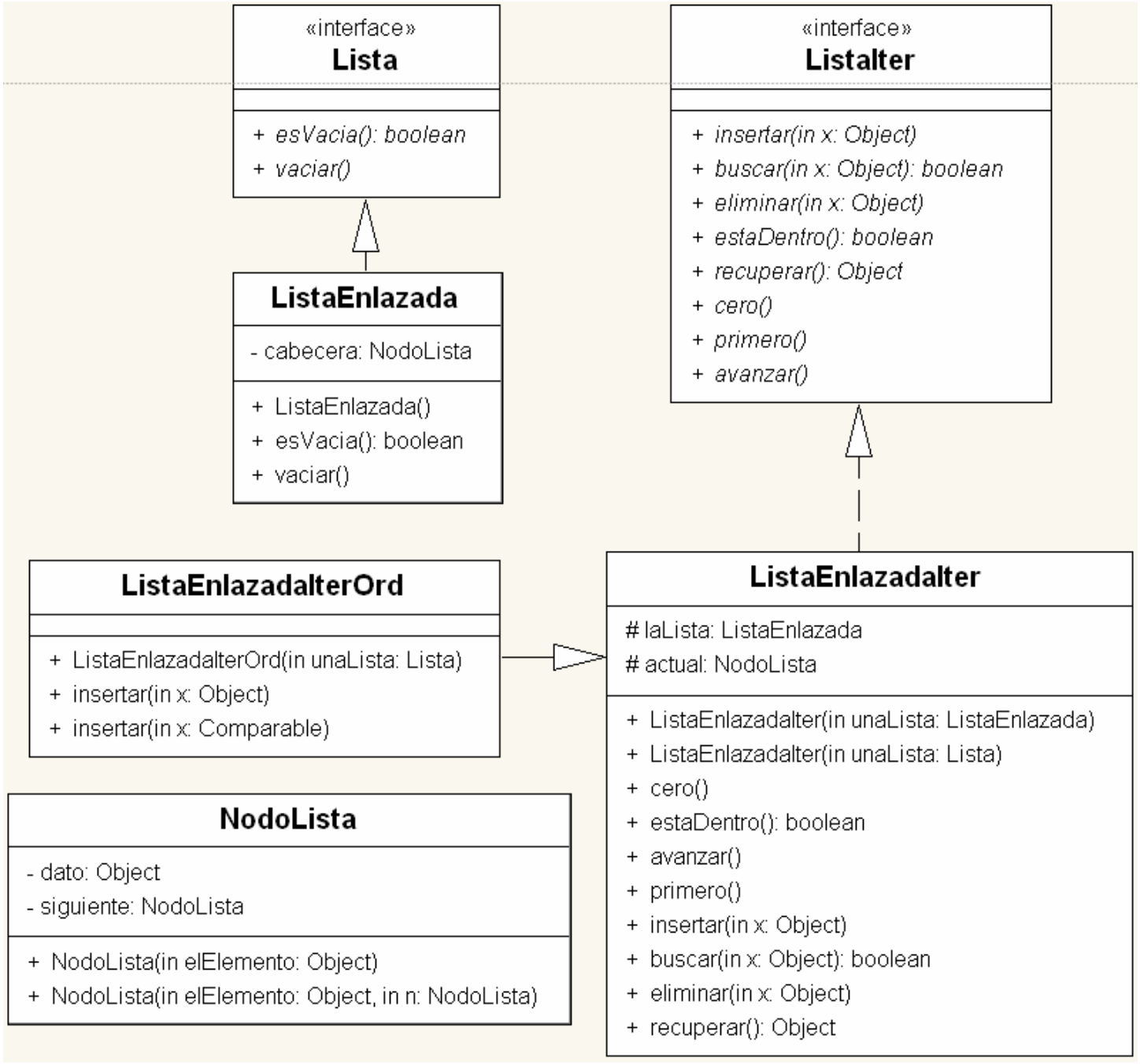
```
public class PilaLinkedList implements Pila {

    private LinkedList lista;

    public PilaLinkedList() {
        lista = new LinkedList();
    }
     * Comprueba si la pila esta vacia

    public boolean esVacia() {
        return lista.isEmpty();
    }
     * Obtiene el elemento mas recientemente insertado en pila
    public Object cima() throws DesbordamientoInferior {
        try {
            return lista.getFirst();
        } catch (Exception e) {
            throw new DesbordamientoInferior("cima");
        }
    }
     * Elimina el elemnto mas recientemente insertado
    public void desapilar() throws DesbordamientoInferior {
        try {
            lista.removeFirst();
        } catch (Exception e) {
            throw new DesbordamientoInferior("desapilar");
        }
    }
     * Devuelve y elimina el elemento mas recientemente inserta en pila
    public Object cimaYDesapilar() throws DesbordamientoInferior {
        try {
            Object temp = lista.getFirst();
            lista.removeFirst();
            return temp;
        } catch (Exception e) {
            throw new DesbordamientoInferior("desapilar");
        }
    }
     * Inserta un elemento nuevo en la pila
    public void apilar(Object x) {
        lista.addFirst(x);
    }
     * pone pila logicamente a vacio
    public void vaciar() {
        lista.clear();
    }
}
```

Colas. Implementación con LinkedList

```
public class ColaLinkedList implements Cola {

    private LinkedList cola;
     * Construye la Cola
    public ColaLinkedList() {

     * Comprueba el estado de la cola
    public boolean esVacia() {
        return cola.isEmpty();
    }

     * Vaciado lógico de la cola
    public void vaciar() {
        cola.clear();
    }

     * Inserta un nuevo elemento en la cola.
    public void insertar(Object x) {
        cola.addLast(x);
    }

     * Devuelve el elemento más antiguo insertado en la cola.
    public Object primero() throws DesbordamientoInferior {
        try {
            return cola.getFirst();
        } catch (NoSuchElementException e) {
            throw new DesbordamientoInferior("Primero");
        }
    }

     * Devuelve y elimina el elemento más antiguo en la cola.
    public Object quitarPrimero() throws DesbordamientoInferior {
        try {
            return cola.removeFirst();
        } catch (NoSuchElementException e) {
            throw new DesbordamientoInferior("Primero");
        }
    }
}
```


«interface»
Cola
+ esVacia(): boolean
+ primero(): Object
+ quitarPrimero(): Object
+ insertar(in x: Object)
+ vaciar()
ColaVec
+ vector: Object
+ tamañoCola: int
+ cabeza: int
+ fin: int
- TAMAÑO_POR_DEFECTO: in
+ ColaVec()
+ esVacia(): boolean
+ vaciar()
+ primero(): Object
+ quitarPrimero(): Object
+ insertar(in x: Object)
ColaLista
- cabeza: NodoLista
- fin: NodoLista
+ ColaLista()
+ esVacia(): boolean
+ vaciar()
+ primero(): Object
+ quitarPrimero(): Object
+ insertar(in x: Object)
ColaLinkedList
- cola: LinkedList
+ ColaLinkedList()
+ esVacia(): boolean
+ vaciar()
+ insertar(in x: Object)
+ primero(): Object
+ quitarPrimero(): Object


«interface»
Pila
+ esVacia()
+ cima()
+ desapilar()
+ cimaYDesapilar()
+ apilar(in x: Object)
+ vaciar()
PilaVec
- vector: Object
- cimaDePila: int
- TAMAÑO: int
+ PilaVec()
+ esVacia()
+ vaciar()
+ cima()
+ desapilar()
+ cimaYDesapilar()
+ apilar(in x: Object)
PilaLista
- cimaDePila: NodoLista
+ PilaLista()
+ esVacia()
+ vaciar()
+ cima()
+ apilar(in x: Object)
+ desapilar()
+ cimaYDesapilar()
PilaLinkedList
- lista: LinkedList
+ PilaLinkedList()
+ esVacia()
+ cima()
+ desapilar()
+ cimaYDesapilar()
+ apilar(in x: Object)
+ vaciar()


«interface»
Lista
+ esVacia(): boolean
+ vaciar()
«interface»
Listalter
+ insertar(in x: Object)
+ buscar(in x: Object): boolean
+ eliminar(in x: Object)
+ estaDentro(): boolean
+ recuperar(): Object
+ cero()
+ primero()
+ avanzar()
ListaEnlazada
- cabecera: NodoLista
+ ListaEnlazada()
+ esVacia(): boolean
+ vaciar()
ListaEnlazadalterOrd
+ ListaEnlazadalterOrd(in unaLista: Lista)
+ insertar(in x: Object)
+ insertar(in x: Comparable)
ListaEnlazadalter
laLista: ListaEnlazada
actual: NodoLista
+ ListaEnlazadalter(in unaLista: ListaEnlazada)
+ ListaEnlazadalter(in unaLista: Lista)
+ cero()
+ estaDentro(): boolean
+ avanzar()
+ primero()
+ insertar(in x: Object)
+ buscar(in x: Object): boolean
+ eliminar(in x: Object)
+ recuperar(): Object
NodoLista
- dato: Object
- siguiente: NodoLista
+ NodoLista(in elElemento: Object)
+ NodoLista(in elElemento: Object, in n: NodoLista)