Estructuras lineales de datos

Esta sección está orientada a comprender y manejar las estructuras lineales de datos, teniendo en cuenta la representación lógica y física de las mismas y el lenguaje de programación utilizado.

CONTENIDOS:

Conceptualización de las estructuras lineales de datos:

- ✓ Introducción y aplicaciones
- ✓ Conceptos básicos, definición y representación lógica
- ✓ Tipo abstracto de dato: Pila, Cola, Dipolo y Lista
- ✓ Tipos de estructuras de almacenamiento

Implantación y análisis de los TAD y diseño de cada primitiva de acceso utilizando la Técnica de Desarrollo de Sistemas de Objetos (TDSO).

- ✓ Pila según los métodos secuencial y el enlazado simple
- ✓ Cola según los métodos secuencial, secuencial circular y enlazado simple.
- ✓ Dipolo según los métodos secuencial y enlazado simple
- ✓ Lista según los métodos: secuencial, enlazado simple, enlazado doble y enlazado doble circular
- ✓ Jerarquía de clases para las estructuras lineales. Una mejor implantación.

El manejo de información relacionada en forma lineal es, hoy por hoy, una de las bases de la mayoría de los paquetes de programación. Existen una infinidad de paquetes programados cuya implementación se centra en el manejo y uso de este tipo de estructura de datos. Es por lo anterior, que pasaremos a continuación a presentar qué son las estructuras lineales de datos y cómo se manipulan en el computador.

1.13. Introducción y aplicaciones

Las estructuras lineales de datos se caracterizan porque sus elementos están en secuencia, relacionados en forma lineal, uno luego del otro. Cada elemento de la estructura puede estar comformado por uno o varios subelementos o campos que pueden pertenecer a cualquier tipo de dato, pero que normalmente son tipos básicos.

Entre las múltiples aplicaciones que tienen estas estructuras podemos mencionar:

* El desarrollo de compiladores de lenguajes de programación que están conformados por varios subprogramas con finalidades más específicas, como por ejemplo: el analizador de lexico que genera la tabla de símbolos.
* La simulación discreta de sistemas a través del computador, donde la mayoría de los paquetes de simulación digital ofrecen lenguajes de simulación que soportan las primitivas para el manejo de colas y sus diferentes versiones.
* La realización de sistemas operativos para los computadores, los cuales hacen un uso intensivo de las estructuras lineales, ya que internamente se soportan en los sistemas operativos, las colas de ejecución para los dispositivos, las pilas de llamadas a los subprogramas de cualquier programa, las listas de usuarios en los sistemas operativos multiusuarios, etc.

1.14. Conceptos básicos y definición

Una estructura lineal de datos o lista está conformada por ninguno, uno o varios elementos que tienen una relación de adyacencia ordenada donde existe un primer elemento, seguido de un segundo elemento y así sucesivamente hasta llegar al último. El tipo de dato de los elementos puede ser cualquiera, pero debe ser el mismo tipo para todos. El valor contenido en los elementos puede ser el mismo o diferente. En estas estructuras se realizan operaciones de agregar y/o eliminar elementos a la lista según un criterio particular. Sobre la base de la forma y el lugar de la realización de estas operaciones en la misma, las listas se clasifican en listas de acceso restringido y listas de acceso no restringido.

Las listas de acceso restringido son las pilas, colas y dipolos. En las pilas, las operaciones de acceso se realizan por un único extremo de la lista, al cual normalmente se denomina **tope** de la pila. En las colas, estas operaciones se realizan por ambos extremos de la lista llamados generalmente, **inicio** y **fin** de la cola. Finalmente, en los dipolos que son colas dobles, las operaciones se realizan también por ambos extremos de la lista, en este caso todas las operaciones se pueden hacer por ambos extremos, es decir se puede insertar o eliminar elementos por el tope o por el fin, a diferencia de la cola donde se inserta siempre por el fin y se elimina por el tope. Se puede entonces considerar al dipolo como una clase general de la clase cola.

Las listas de acceso no restringido, denominadas listas, son el tipo más general, al cual se considera como la superclase de las otras clases de listas, en específico de las pilas, colas y dipolos. Haciendo la jerarquía de clases adecuada para estas estructuras, se tiene que la lista es la clase raíz de la jerarquía que tiene como subclases la pila, la cola y el dipolo, este último a su vez tiene como subclases el dipolo de entrada restringida y el dipolo de salida restringida. Esta jerarquía se observa en la figura 1.23.

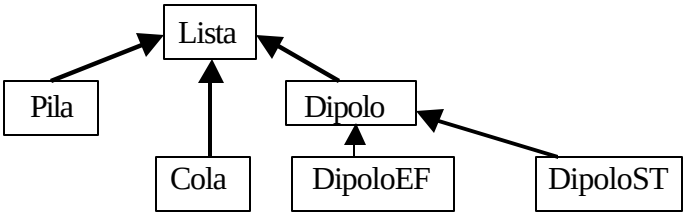


Figura 1.23. Jerarquía de clases para las listas.

Una lista es una secuencia de cero o más elementos de un mismo tipo.

Cada elemento de la lista tiene asignado un tipo de dato. Si **l** es del tipo ListaDe [TipoEle: e] entonces e_1, e_2, ⋯ , e_n conforman la lista **l** cuyos elementos tienen asignado un mismo tipo.

Las propiedades de las listas son:

1) Si n = 0 entonces la lista está vacía
2) Si $n \geq 1$ entonces e_1 es el primer elemento de la lista y e_n el último.
3) e_i es el predecesor de e_{i+1} y el sucesor de e_{i-1} con $1 \leq i \leq n$

Ejemplo: Sea la lista l = (‘casa’, ‘perro’, ‘carro’, ‘árbol’) entonces
‘casa’ es el primer elemento de l y **no** tiene predecesor

'árbol' es el último elemento de l y **no** tiene sucesor
'casa' es el predecesor de 'perro'
'perro' es el sucesor de 'casa'

Las listas se pueden clasificar por varios criterios:

Por el orden de sus elementos sobre la base de un subelemento: ordenadas (ascendente o descendente), y desordenadas.

Por el método de almacenamiento: secuencial y enlazada (simple, doble, simple circular y doble circular).

1.15. Representación lógica de las estructuras lineales de datos

Las listas se representan en forma lógica como una secuencia de elementos, por ejemplo: una lista de los estudiantes inscritos en un curso, una lista de productos de un supermercado, etc. En forma gráfica se tiene:

Harina de maíz PAN	12
Harina de trigo Robin Hood	34
Café Brasil	50
Margarina Mavesa	28

Una lista se representa lógicamente como: e_1 | | e_2 | ··· | e_n | y cada tipo de lista tiene la misma representación.

1.16. Especificación de los tipos abstractos de datos

Cada uno de los tipos de lista tiene su propia definición y serán representados como TAD. Un TAD se define con dos partes: una especificación y una o varias implementaciones.

Todas las operaciones que se definen para las listas involucran un solo elemento, o sea al agregar o insertar, se incluye un elemento en la lista, igual al eliminar o suprimir.

A continuación se presentan cada uno de los tipos de lista y su especificación como TAD.

1.16.1. Pila

Una pila es un subtipo de las listas donde el acceso está restringido a un solo extremos de la lista, en este caso al tope de la misma. Un ejemplo de esta estructura es una pila de bandejas de un restaurante de comida rápida (*self service*) o una pila de platos, etc. Si se analiza el ejemplo en detalle, se tiene que cualquier cliente del restaurante, al llegar toma la primera bandeja que encuentra en la pila de bandejas, es decir la que está encima de todas las demás. Asimismo, cuando el empleado del restaurante coloca bandejas limpias en la pila, lo hace colocándolas encima de la que está arriba, puesto que es más trabajo, alzar algunas y colocar las limpias entre las que quedan y las que alzó.

Las operaciones básicas sobre una pila son: crearla, destruirla, agregar un nuevo elemento, suprimir un elemento, consultar el elemento del tope y verificar si está vacía. Sobre la base de estas operaciones se especifíca el TAD Pila como se muestra en la figura 1.24. Esta especificación incluye operaciones que pueden ser extendidas en la implementación para soportar otras operaciones útiles de acuerdo a las aplicaciones que la puedan utilizar. En

particular, toda implementación debe contener las operaciones básicas definidas para el tipo y puede ser ampliada con otras adicionales.

12/3/98 Especificación Pila[TipoEle]		
1	**Especificación sintáctica:** creaPila()⟶ Pila, meterElePila(Pila,TipoEle)⟶ Pila, sacarElePila(Pila)⟶ Pila, conTopePila(Pila)⟶ TipoEle, vacíaPila(Pila)⟶ Lógico, destruyePila(Pila)⟶ .	-**creaPila()**: Crea una pila vacía. -**meterElePila()**: Ingresa un nuevo elemento a la pila por el tope de la misma. -**sacarElePila()**: Elimina el elemento que está actualmente en el tope de la pila. Si la pila está vacía no hace ninguna eliminación. -**vacíaPila()**: Regresa Verdadero si la pila está vacía. -**destruyePila()**: Destruye la pila. -**conTopePila()**: Devuelve el elemento que se encuentra actualmente en el tope de la pila. Si la pila está vacía devuelve un valor especial que lo indica.
2	**Declaraciones:** TipoEle: e, {TipoEleNoDef}	
3	**Especificación semántica:** vacíaPila(creaPila())=Verdadero vacíaPila(meterElePila(creaPila(),e))=Falso conTopePila(creaPila())={TipoEleNoDef} sacarElePila(creaPila())=creaPila()	

Figura 1.24. Especificación del TAD Pila en TDSO.

1.16.2. Cola

Una cola es otro subtipo de las listas donde el acceso está restringido a los extremos de la lista, es decir al inicio y al fin de la misma. Un ejemplo de esta estructura es una cola de personas en un restaurante de comida rápida (*self service*). En el ejemplo, se tiene que cualquier cliente del restaurante, al llegar entra a la fila de clientes que van a comer en el mismo por el final de la fila. Si la cola tiene algunos clientes, él debe esperar hasta llegar al inicio de la cola para poder ser servido. El cliente al inicio de la cola sale de la misma, en cuanto deja de ser servido, es decir ya tiene su comida y pasa a una mesa disponible para comer. Al salir de la cola el primer cliente, los que están detrás se mueven o lo que es lo mismo, el inicio o tope de la cola se mueve al que era el segundo cliente que pasa a ser el nuevo primer cliente en la cola. Asimismo, cuando el nuevo cliente llega a la cola, se coloca al final de la misma, por lo cual el fin de la cola se mueve a la posición del nuevo cliente.

Aunque, teóricamente, cualquier cliente puede salirse de la cola en un momento dado, en este caso no se le permite, la única manera de salir es llegar a la posición inicial. Se puede hacer la analogía con una fila donde hay barandas de lado y lado, que normalmente colocan para evitar que las personas entren y salgan por otro lugar que no sea el inicio y fin de la fila.

Similar al subtipo pila, en la cola las operaciones básicas son: creación, destrucción, inserción al final de un nuevo elemento, eliminación del inicio de un elemento, consultar que elemento está al inicio y cual al final, y verificar si la cola está vacía. Según estas operaciones se especifica el TAD Cola que se muestra en la figura 1.25.

27/3/98 Especificación Cola[TipoEle]		
1	**Especificación sintáctica:** creaCola()⟶ Cola, entrarCola(Cola,TipoEle)⟶ Cola, salirCola(Cola)⟶ Cola, conEleCola(Cola)⟶ TipoEle, vacíaCola(Cola)⟶ Lógico, destruyeCola(Cola)⟶ .	-**creaCola()**: Crea una cola vacía. -**entrarCola()**: Ingresa un nuevo elemento a la cola por el fin de la misma. -**salirCola()**: Elimina el elemento que está actualmente en el inicio de la cola. Si la cola está vacía no hace ninguna eliminación. -**destruyeCola()**: Destruye la cola. -**vacíaCola()**: Regresa Verdadero si la cola está vacía. -**conEleCola()**: Devuelve el elemento que se encuentra actualmente en el inicio de la cola. Si la cola está vacía devuelve un valor especial que lo indica.
2	**Declaraciones:** TipoEle: e, {TipoEleNoDef}	
3	**Especificación semántica:** vacíaCola(creaCola())=Verdadero vacíaCola(meterEleCola(creaCola(),e))=Falso conEleCola(creaCola())={TipoEleNoDef} sacarEleCola(creaCola())=creaCola()	

Figura 1.25. Especificación del TAD Cola en TDSO.

1.16.3. Dipolo

Esta estructura equivale a dos colas colocadas una en un sentido y la otra en sentido contrario, por ello las operaciones de inserción y eliminación se pueden realizar por ambos extremos. Dos casos especiales se pueden tener, el dipolo de entrada restringida donde sólo se puede insertar por un extremo y eliminar por ambos, y el dipolo de salida restringida, donde se puede insertar por ambos extremos y sólo se puede suprimir por un extremo. Se llamará a estos extremos como izquierdo (izq) y derecho (der).

La figura 1.26 presenta la especificación de este subtipo de lista. Sus operaciones básicas son: creación, destrucción, verificación de dipolo vacío, inserción de un nuevo elemento por la izquierda, inserción por la derecha, eliminación por la izquierda, eliminación por la derecha, consulta del elemento que está más a la izquierda y del que está más a la derecha.

1.16.4. Lista

La lista es el tipo más general de estructura lineal donde las inserciones y eliminaciones se hacen en cualquier punto de la lista, por ello se debe especificar donde se requiere que se haga la operación. La especificación de este TAD se muestra en la figura 1.27. Sus operaciones básicas son: creación, destrucción, inserción, eliminación, consulta y verificación de lista vacía.

27/3/98 Especificación Dipolo[TipoEle]		
1	**Especificación sintáctica:** creaDipolo()⟶ Dipolo, insIzqDipolo(Dipolo,TipoEle)⟶ Dipolo, insDerDipolo(Dipolo,TipoEle)⟶ Dipolo, eliIzqDipolo(Dipolo)⟶ Dipolo, eliDerDipolo(Dipolo)⟶ Dipolo, conIzqDipolo(Dipolo)⟶ TipoEle, conDerDipolo(Dipolo)⟶ TipoEle, vacíoDipolo(Dipolo)⟶ Lógico, destruyeDipolo(Dipolo)⟶ .	-**creaDipolo()**: Crea un Dipolo vacío. -**insIzqDipolo(), insDerDipolo()**: Ingresa un nuevo elemento al Dipolo por la izquierda o por la derecha del mismo, respectivamente. -**eliIzqDipolo(), eliDerDipolo()**: Elimina el elemento actual a la izquierda o a la derecha del Dipolo. Si el Dipolo está vacío no hace ninguna eliminación. -**destruyeDipolo()**: Destruye el Dipolo. -**vacíoDipolo()**: Regresa Verdadero si el Dipolo está vacío. -**conIzqDipolo(), conDerDipolo()**: Devuelven el elemento actual a la izquierda y a la derecha del Dipolo, respectivamente. Si el Dipolo está vacío, devuelve un valor especial que lo indica.
2	**Declaraciones:** TipoEle: e, {TipoEleNoDef}	
3	**Especificación semántica:** vacíoDipolo(creaDipolo())=Verdadero vacíoDipolo(insIzqDipolo(creaDipolo(),e))=F vacíoDipolo(insDerDipolo(creaDipolo(),e))=F conIzqDipolo(creaDipolo())={TipoEleNoDef} conDerDipolo(creaDipolo())={TipoEleNoDef} eliIzqDipolo(creaDipolo())=creaDipolo() eliDerDipolo(creaDipolo())=creaDipolo()	

Figura 1.26. Especificación del TAD Dipolo en TDSO.

27/3/98 Especificación Lista[TipoEle]		
1	**Especificación sintáctica:** creaLista()⟶ Lista, insLista(Lista,TipoEle)⟶ Lista, eliLista(Lista)⟶ Lista, conLista(Lista)⟶ TipoEle, vacíaLista(Lista)⟶ Lógico, destruyeLista(Lista)⟶ .	-**creaLista()**: Crea una lista vacía. -**insLista()**: Ingresa un nuevo elemento a la lista según una posición especificada. -**eliLista()**: Elimina el elemento en la lista según una posición especificada. Si la lista está vacía no hace ninguna eliminación. -**destruyeLista()**: Destruye la lista. -**vacíaLista()**: Regresa Verdadero si la lista está vacía. -**conLista()**: Devuelve el elemento en la lista seguún una posición especificada. Si la lista está vacía devuelve un valor especial que lo indica.
2	**Declaraciones:** TipoEle: e, {TipoEleNoDef}	
3	**Especificación semántica:** vacíaLista(creaLista())=Verdadero vacíaLista(insLista(creaLista(),e))=Falso conLista(creaLista())={TipoEleNoDef} eliLista(creaLista())=creaLista()	

Figura 1.27. Especificación del TAD Lista en TDSO.

1.17. Estructuras de almacenamiento para las estructuras lineales de datos

Para almacenar en memoria una lista se utilizan dos métodos: el secuencial y el enlazado. En el primero los elementos de la lista están contiguos físicamente en la memoria y para su soporte se utiliza el tipo Arreglo, teniendo la ventaja de efectuar los accesos a los elementos

en forma directa, solamente indicando el subíndice o posición del elemento en la estructura. Por ello, el acceso es más rápido pero con la desventaja de restricción del crecimiento de la lista que está sujeta al tamaño máximo del arreglo. Así también, esto conlleva a otra desventaja que es la pérdida de espacio usado cuando la lista no ocupa más de la mitad del espacio supuesto para la lista. Ambas desventajas tienen su origen en que los arreglos se declaran o crean con un tamaño máximo fijo.

La representación física de las listas según este método está ilustrada en la figura 1.28.

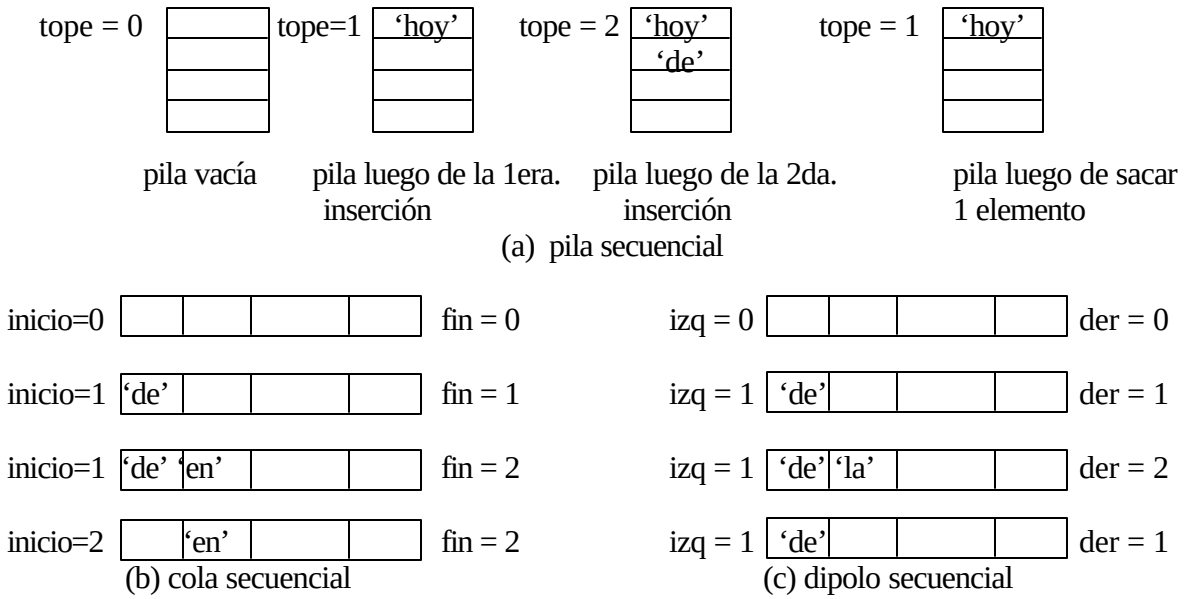


Figura 1.28. Representación física de pila, cola y dipolo.

En la parte (a) de la figura se observa una pila en varios estados posibles, vacía, con un único elemento luego de la primera inserción del elemento cuyo valor es 'hoy', luego de una segunda inserción de un nuevo elemento que contiene 'de' y por último, luego de la primera eliminación donde la pila sólo contiene el valor que se insertó de primero.

En la parte (b) de la misma figura se muestra una cola inicialmente vacía, donde inicio = fin = 0. Luego, una primera inserción, siempre por el fin, del elemento que contiene el valor 'de', por lo cual ese elemento pasa a ser el inicio y el fin de la cola. A continuación, se inserta el elemento que contiene 'en' y él pasa a ser el nuevo fin de la cola. Por último, una eliminación, siempre por el inicio de la cola, por ello el inicio se mueve a la posición del nuevo inicio que es el elemento 'en'. Se puede observar que luego de esto **es imposible** reusar la posición 1 del arreglo, a menos que la cola se vacíe, pues una nueva inserción siempre será hecha por el fin de la cola y **nunca** por el inicio.

Un dipolo secuencial se puede observar en la figura 1.28. (c) que inicialmente está vacío y luego de la primera inserción, ambos indicadores, izq y der, se colocan en uno para indicar que sólo hay un elemento en el dipolo, esta inserción arroja el mismo resultado si fue hecha por la izquierda o por la derecha. A continuación, se inserta un nuevo elemento de contenido 'la' que fue insertado por el extremo denominado der, puesto que por la izquierda ya no hay espacio para inserciones. Luego se efectúa una eliminación, que fue hecha por la derecha del

dipolo, puesto que si hubiera sido hecha por la izquierda, se habría eliminado el elemento 'de' en vez del elemento 'la'. Se puede observar con esto, que en el caso del dipolo si es posible reusar las posiciones del dipolo, probando por cual extremo hay espacio disponible.

1.18. Implantación de los TAD y diseño de cada primitiva de acceso utilizando la Técnica de Desarrollo de Sistemas de Objetos (TDSO).

Las implementaciones de los tipos de datos dependen del método de representación física que se seleccione para la estructura. A continuación se incluyen algunas implementaciones posibles para los TAD de esta unidad, ellas son las que más se usan en las aplicaciones. De le daja al lector, como ejercicio, la implementación de las mismas con versiones de los métodos no incluidas en este trabajo.

1.18.1. TAD: Pila según los métodos secuencial y el enlazado simple

Pila secuencial:

La figura 1.29 muestra la implementación de una pila según el método secuencial, donde se utiliza un arreglo de tamaño fijo, que en este caso se escogió 100, pero este valor puede variar dependiendo de la aplicación para la cual se necesita usar la pila. Lo importante aquí es resaltar que en el caso del método secuencial se usa una asignación de memoria contigua y fija de antemano. En el caso que esto no sea lo indicado para la aplicación, se recomienda utilizar el método enlazado.

El análisis de complejidad en tiempo para las funciones de esta implementación se hace de igual manera que la hecha para la clase Cadena, por lo cual el orden de cada función será colocado junto con cada nombre de función. La figura 1.30 presenta el constructor.

PilaSec[TipoEle] Clases: Entero, Lógico, TipoEle, ArregloDe TipoEle		
1	**Superclases:** ninguna	-**max**: Número máximo de elementos.
2	**Estructura**: privado: max : Entero+ = 100 tope : Entero+ = 0 p : Arreglo[100]De TipoEle	-**tope**: Número actual de elementos. -**p**: Arreglo que contiene los elementos actuales en la pila. -**PilaSec()**. *Constructor.* Crea una pila vacía.
3	**Operaciones:** público: PilaSec() meterElePila(TipoEle: e) sacarElePila() conPila(): TipoEle vacíaPila(): Lógico numEle(): Entero+ despliegue()	-**meterElePila()**. *Transformador.* Ingresa un nuevo elemento a la pila. -**sacarElePila()**. *Transformador.* Elimina el elemento que está actualmente en el tope de la pila, si no está vacía. -**conPila()**. *Observador.* Devuelve el elemento que se encuentra actualmente en el tope de la pila, si existe. -**vacíaPila()**. *Observador.* Regresa Verdadero si la pila está vacía, de lo contrario regresa Falso. -**numEle()**. *Observador.* Regresa el tope. -**despliegue().** *Observador.* Despliega el contenido de la pila.

Figura 1.29. Implementación según el método secuencial.

12/3/98 PilaSec() {pos: tope = 0 ∧ max = 100}		
1	max, tope = 100, 0	**-max, tope.** Definidos en PilaSec.
1	PilaSec[Entero] pila1 ⇒ pila1.max = 100 y pila1.tope = 0	Crea la variable pila1 de enteros.

Figura 1.30. Constructor vacío. O(1).

Las figuras de la 1.31 a la 1.34 presentan la verificación de pila vacía, el observador sobre el tamaño actual de la pila, la que despliega en pantalla el contenido de la pila y la que inserta un nuevo elemento en la pila, respectivamente.

12/3/98		
	vacíaPila(): Lógico	
{pre: tope ≥ 0 ∧ max = 100}		{pos: tope ≥ 0 ∧ max = 100}
1	regresa (tope = 0)	**-tope.** Definido en PilaSec
1	pila1.vacíaPila() ⇒ 1	La variable pila1 está vacía

Figura 1.31. Verifica si la pila está vacía. O(1).

12/3/98		
	numEle(): Entero+	
{pre: tope ≥ 0 ∧ max = 100}		{pos: tope ≥ 0 ∧ max = 100}
1	regresa tope	**-tope.** Definido en PilaSec
1	pila1.numEle() ⇒ 0	La variable pila1 tiene 0 elementos

Figura 1.32. Devuelve el número de elementos actuales en la pila. O(1).

12/3/98		
	despliegue ()	
{pre: tope ≥ 0 ∧ max = 100}		{pos: tope ≥ 0 ∧ max = 100}
1 2	Despliegue “Elementos de la pila en orden inverso” [Despliegue “Elemento(“, i , “)=” , este → p (i)] i = 0, tope –1	**-i**: Entero+. Subíndice del lazo. **-este**: Apuntador al objeto actual.
1	pila1=(100, 2, 45, 34) ⇒ Elementos de la pila en orden inverso / Elemento(0)=45 / Elemento(1)=34 /	Despliega el contenido de la pila1 con sus comentarios

Figura 1.33. Despliega los elementos que están en la pila. O(n).

12/3/98		
	meterElePila(TipoEle: e)	
{pre: e ≠ {TipoEleNoDef} ∧ 0 ≤ tope < max }		{pos: 0 ≤ tope < max }
1	Si (tope = max) entonces Despliegue “Pila llena” sino p (tope) = e tope = tope + 1 fsi	**-e**: TipoEle. Nuevo elemento para ser insertado en la pila. - **tope, max, p**: Definidos en PilaSec.
1 2	pila1=(100, 2, 45, 34) y e = 8 ⇒ pila1=(100, 3, 45, 34, 8) pila1=(100,100, 45, ...) y e = 8 ⇒ pila1=(100,100, 45, ...) / Pila llena	Inserta 8 en pila1 No inserta 8 pues la pila está llena

Figura 1.34. Ingresa un nuevo elemento en la pila. O(1).

La eliminación de un elemento en la pila y la consulta sobre el elemento en el tope de la pila se muestran en las figuras 1.35 y 1.36, respectivamente.

12/3/98	sacarElePila()	
{pre: 0 ≤ tope < max }		{pos: 0 ≤ tope < max }

1	Si (tope = 0) entonces
	  Despliegue "Pila vacía"
	Sino
	  tope = tope – 1
	fsi
		- **tope, max, p**: Definidos en PilaSec.

1 2	pila1=(100, 2, 45, 34) ⇒ pila1=(100, 3, 45) pila1=(100, 0) ⇒ Pila vacía	Elimina el elemento en el tope No elimina pues la pila está vacía

Figura 1.35. Elimina el elemento en el tope de la pila. O(1).

12/3/98	conPila(): TipoEle	
{pre: 0 ≤ tope < max }		{pos: 0 ≤ tope < max }

1	Si (tope = 0) entonces
	  regrese {TipoEleNoDef}
	sino
	  regrese p (tope)
	fsi
		-tope, max, p: Definidos en PilaSec.
		-{TipoEleNoDef}: Definido en meterElePila().

1 2	pila1=(100, 2, 45, 34) ⇒ 34 pila1=(100, 0) ⇒ {TipoEleNoDef}	Se obtiene una copia del elemento en el tope Se obtiene el elemento especial

Figura 1.36. Regresa el valor del elemento en el tope de la pila. O(1).

Pila enlazada simple:

La implementación de una pila según el método enlazado simple presentada en la figura 1.37 amerita la implementación de una clase adicional que es la clase Nodo[TipoEle], ella debe ofrecer las operaciones básicas que son: al menos un constructor, el destructor y los observadores y transformadores para cada uno de los componentes del nodo. En este caso, se tomarán solamente los campos **info** y **pSig**, que significan la información que se almacenará en la pila que es de tipo TipoEle y el apuntador al siguiente nodo en la pila, respectivamente. La figura 1.38 presenta un ejemplo de la implementación de la clase Nodo[TipoEle].

PilaEnl[TipoEle] Clases: Entero, Lógico, ApuntadorA, TipoEle, Nodo[TipoEle]		
1	**Superclases:** ninguna	**-n**. Número actual de elementos.
2	**Estructura**: privado: n : Entero+ = 0 pTope : ApuntadorA Nodo[TipoEle] = Nulo	**-pTope.** Dirección del nodo tope. **-PilaEnl()**. *Constructor.* Crea una pila vacía. **-~PilaEnl():** *Destructor.* Destruye la pila.
3	**Operaciones:** público: PilaEnl() ~PilaEnl() meterElePila(TipoEle: e) sacarElePila() conPila(): TipoEle vacíaPila(): Lógico numEle(): Entero+ despliegue()	**-meterElePila()**. *Transformador.* Ingresa un nuevo elemento a la pila. **-sacarElePila()**. *Transformador.* Elimina el elemento que está actualmente en el tope de la pila, si no está vacía. **-conPila()**. *Observador.* Devuelve el elemento que se encuentra actualmente en el tope de la pila, si existe. **-vacíaPila()**. *Observador.* Regresa Verdadero si la pila está vacía. **-numEle()**. Observador. Regresa el tope. **-despliegue()**. Observador. Despliega el contenido de la pila.

Figura 1.37. Implementación de una pila según el método enlazado simple.

Nodo[TipoEle]		
1	**Superclases:** ninguna	-**info**: Información del nodo.
2	**Estructura**: privado: info: TipoEle = {TipoEleNoDef} pSig : ApuntadorA Nodo[TipoEle] = Nulo	-**pSig**: Dirección del nodo siguiente. -**Nodo()**. *Constructor.* Crea un nodo con un elemento no definido y un apuntador nulo.
3	**Operaciones:** público: Nodo() Info(TipoEle: e) Info(): TipoEle PSig(ApuntadorA Nodo[TipoEle]: pn) PSig(): ApuntadorA Nodo[TipoEle]	-**Info()**. *Transformador.* Modifica la información del nodo por la que viene en el parámetro. -**PSig()**. *Transformador.* Modifica el apuntador del nodo por la dirección que viene en el parámetro. -**Info()**. *Observador.* Devuelve la información contenida en el nodo. -**PSig()**. *Observador.* Regresa la dirección del nodo siguiente contenida en el nodo.

Figura 1.38. Implementación de la clase Nodo[TipoEle].

En este caso se usa memoria dispersa, por lo cual un estado posible de la pila enlazada sería el mostrado en la figura 1.39, donde la variable **p1** está declarada del tipo PilaEnl[TipoEle] y luego de tres inserciones de los elementos 'la', 'de' y 'al'.

A continuación se presentan las figuras desde la 1.40 hasta la 1.47, que contienen las funciones de la clase pila enlazada junto con su orden de complejidad.

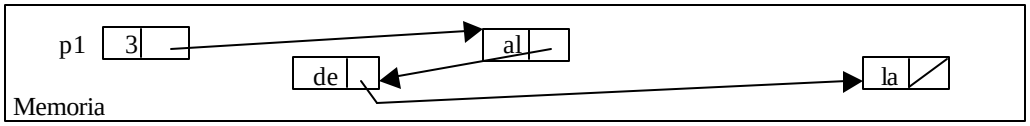


Figura 1.39. Almacenamiento de una pila enlazada.

12/3/98 PilaEnl() {pos: n = 0 ∧ pTope = Nulo}		
1	n, pTope = 0, Nulo	**-n, pTope.** Definidos en Pila.
1	PilaEnl[Entero] pila1 ⇒ pila1.n = 0 y pila1.pTope = Nulo	Crea la variable pila1 de enteros.

Figura 1.40. Constructor vacío. O(1).

12/3/98 vacíaPila(): Lógico {pre: n ≥ 0}		{pos: n ≥ 0}
1	Regresa (pTope = Nulo)	**-pTope.** Definido en Pila
1	pila1.vacíaPila() ⇒ Verdadero	La variable pila1 está vacía

Figura 1.41. Verifica si la pila está vacía. O(1).

12/3/98 numEle(): Entero+ {pre: n ≥ 0}		{pos: n ≥ 0}
1	Regresa n	**-n.** Definido en Pila
1	pila1.numEle() ⇒ 0	La variable pila1 tiene 0 elementos

Figura 1.42. Devuelve el número de elementos actuales en la pila. O(1).

12/3/98 despliegue () {pre: n ≥ 0}		{pos: n ≥ 0}
1 2 3	Despliegue “Elementos de la pila en orden inverso” pAux = pTope [Despliegue “Elemento(“, i , “)=” , Paux → Info() pAux = pAux → PSig()] i = 1, n	**-pAux:** ApuntadorA Nodo[TipoEle]. Variable auxiliar para recorrer la pila. **-Info(), PSig():** Definidos en la clase Nodo. **-i**: Entero+. Subíndice del lazo.
1	pila1=(2, →) (45, →) (34, /) ⇒ Elementos de la pila en orden inverso / Elemento(0)=45 / Elemento(1)=34 /	Despliega el contenido de la pila1 con sus comentarios

Figura 1.43. Despliega los elementos actuales en la pila. O(n).

12/3/98 meterElePila (TipoEle: e) {pre: e ≠ {TipoEleNoDef} ∧ n ≥ 0}		{pos: n ≥ 0}
1 2	pAux = nuevaDir Nodo Si (pAux ≠ Nulo) entonces pAux → Info(e) pAux → PSig(pTope) pTope = pAux n = n + 1 sino Despliegue “Memoria llena” fsi	**-pAux:** ApuntadorA Nodo[TipoEle]. Variable auxiliar. **-Info(), PSig():** Definidos en la clase Nodo. **-n, pTope**: Definidos en Pila. **-nuevaDir Nodo:** Regresa una dirección de memoria disponible de tipo Nodo.
1	pila1=(2, →) (45, →) (34, /), e = 20 ⇒ pila1 = (3, →) (20, →) (45, →) (34, /)	Ingresa 20 a la pila1

Figura 1.44. Inserta un nuevo elemento en la pila. O(1).

12/3/98 sacarElePila () {pre: n ≥ 0}		{pos: n ≥ 0}
1	Si (pTope ≠ Nulo) entonces pAux = pTope pTope = pTope → PSig( ) devuelvaDir pAux n = n - 1 sino Despliegue “Pila vacía” fsi	**-pAux:** ApuntadorA Nodo[TipoEle]. Variable auxiliar. **-PSig():** Definido en la clase Nodo. **-n, pTope**: Definidos en Pila. **-devuelvaDir:** Regresa la dirección a la memoria disponible.
1	pila1=(0, /) ⇒ Pila vacía	No elimina
2	pila1=(2, →) (45, →) (34, /) ⇒ pila1 = (1, →) (34, /)	Elimina 45 de la pila1

Figura 1.45. Elimina el elemento en el tope de la pila. O(1).

La complejidad de la clase Pila implementada con el método enlazado es O(n), ya que las funciones **despliegue()** y el destructor tienen ese orden y éste domina el orden constante de las demás funciones.

12/3/98 conPila (): TipoEle		
{pre: n ≥ 0}		{pos: n ≥ 0}
1	Si (pTope = Nulo) entonces regrese {TipoEleNoDef} sino regrese pTope → Info() fsi	**-Info():** Definido en la clase Nodo. **-n, pTope**: Definidos en Pila.
1	pila1=(0, /) ⇒ {TipoEleNoDef}	Devuelve el valor {TipoEleNoDef}
2	pila1=(2, →) (45, →) (34, /) ⇒ 45	Devuelve 45

Figura 1.46. Devuelve el contenido del tope de la pila. O(1).

12/3/98 ~PilaEnl()		
{pre: n ≥ 0}		
1	(pTope ≠ Nulo) [pAux = pTope pTope = pTope → PSig() devuelvaDir pAux]	**-pTope.** Definido en Pila. **-PSig():** Definido en la clase Nodo. **-pAux:** ApuntadorA Nodo[TipoEle]. Variable auxiliar

Figura 1.47. Destruye la pila enlazada. O(n).

1.18.2. TAD: Cola según los métodos secuencial, enlazado y secuencial circular.

Cola secuencial:

La figura 1.48 muestra la implementación de una cola según el método secuencial, donde se utiliza un arreglo de tamaño fijo, en este caso se escogió 100, pero este valor puede variar dependiendo de la aplicación para la cual se necesita usar la cola. En el caso del método secuencial se usa una asignación de memoria contigua y fija de antemano. Si esto no es lo indicado para la aplicación, se recomienda utilizar el método enlazado.

El análisis de complejidad en tiempo para las funciones de esta implementación se hace de igual manera que la hecha para la clase Pila.

ColaSec[TipoEle] Clases: Entero, Lógico, TipoEle, ArregloDe		
1	**Superclases:** ninguna	**-inicio**: Subíndice del primer elemento.
2	**Estructura:** privado inicio: Entero+ = 0 fin: Entero+ = 0 max: Entero+ = 100 c : Arreglo[100]De TipoEle	**-fin**: Subíndice del último elemento. **-max:** Número máximo de elementos **-c**: Arreglo que contiene los elementos de la cola. **-ColaSec()**. *Constructor.* Crea una cola vacía.
3	**Operaciones:** público ColaSec(): entrarCola(TipoEle: e) salirCola() conCola(): TipoEle vaciaCola(): Lógico numEle(): Entero+ despliegue()	**-entrarCola()**. *Transformador.* Ingresa un nuevo elemento por el fin de la cola. **-salirCola()**.*Transformador.* Elimina el elemento que se encuentra en el inicio de la cola, si la cola no está vacía. **-conCola()**. *Observador.* Devuelve el elemento que se encuentra actualmente en el inicio de la cola, si existe **-vacíaCola()**. *Observador.* Regresa verdadero si la cola está vacía. **-numEle()**. *Observador.* Regresa el número actual de elementos, si no está vacía. **-despliegue()**. *Observador.* Despliega el contenido actual de la cola.

Figura 1.48. Implementación de la clase cola secuencial.

El constructor vacío de esta clase se puede observar en la figura 1.49, el cual crea e incia las variables de este tipo.

23/3/98 ColaSec() {pos: inicio = 0 ∧ fin = 0 ∧ max =100}		
1	inicio, fin, max = 0, 0, 100	**-inicio, fin, max:** Definidos en ColaSec.
1	ColaSec[Entero] cola1 ⇒ cola1.max = 100, cola1.inicio = 0, cola1.fin = 0	Crea la variable cola1 de enteros.

Figura 1.49. Constructor vacío. O(1).

Para ingresar un nuevo elemento a la cola se utiliza una implementación como la de la figura 1.50. Este nuevo elemento se coloca siempre al final de la cola, es decir esta cola sigue la política primero en entrar primero en salir, cuyas siglas en inglés es FIFO.

23/3/98 entrarCola(TipoEle: e) {pre: e ≠ {TipoEleNoDef} ∧ 0 ≤ fin < max} { pos: C´= C + {e} ∧ 0 ≤ fin ≤ max }		
1	Si (fin = max) entonces Despliegue "Cola llena" sino c (fin) = e fin = fin + 1 fsi	**-e:** TipoEle. Nuevo elemento a ser insertado. **-inicio, fin, max:** Definidos en ColaSec.
1	cola1 = (0, 2, 100, 71, 4),e = 3 ⇒ cola1 = (0,3,71,4,3)	Inserta 3 en la cola.
2	cola1=(5,100,100,7,..), e =8 ⇒ cola1= (5,100,100,7,..)	No hace la inserción, la cola está llena.

Figura 1.50. Inserción de un nuevo elemento en la cola secuencial. O(1).

La eliminación o retiro de elementos de la cola secuencial es hecha como lo muestra la figura 1.51 y se hace siempre por el inicio de la misma.

23/3/98 salirCola() { pre: 0 ≤ inicio < max } {pos: C` = C – C(inicio) ∧ 0 ≤ inicio ≤ max }		
1	Si (inicio = 0 ∧ fin = 0) entonces Despliegue "Cola vacía" sino inicio = Si (inicio = fin – 1) entonces fin = 0 0 sino inicio + 1 fsi fsi	**-inicio, fin, max:** Definidos en ColaSec.
1	cola1= (0, 2, 100, 71, 4) ⇒ cola1= (1, 2, 100, 71, 4)	Elimina el elemento en inicio
2	cola1= (1, 3, 100, 71, 4, 32) ⇒ cola1(2,3,100,71,4,32)	Elimina el elemento en inicio
3	cola1= (0, 0, 100) ⇒ cola1(0, 0, 100)	No elimina, cola vacía
4	cola1= (0, 1, 100, 23) ⇒ cola1(0, 0, 100)	Elimina el único elemento de la cola

Figura 1.51. Eliminación del elemento al inicio de la cola. O(1).

La consulta del elemento en el inicio de la cola o el primero de la misma se hace con la función descrita en la figura 1.52, ella regresará la copia del elemento si la cola no está vacía.

23/3/98	ConCola():TipoEle	
{pre: $0 \leq$ inicio < max}		{pos: inicio $\geq 0 \wedge$ fin $\geq 0 \wedge$ max = 100 }
1	Si (inicio = $0 \wedge$ fin = 0) entonces regrese {TipoEleNoDef} sino regrese c (inicio) fsi	**-inicio, fin, c.** Definidos en ColaSec.
1 2 3	cola1 = (0, 1, 100, 71, 4) $\Rightarrow$ 71 cola1 = (1, 2, 100, 71, 4, 32) $\Rightarrow$ 4 cola1 = (0, 0, 100) $\Rightarrow$ -200	Se obtiene una copia del elemento en el inicio Se obtiene una copia del elemento en el inicio Se obtiene el elemento especial = -200

Figura 1.52. Consulta del elemento al inicio de la cola secuencial. O(1).

De las figuras 1.53 a la 1.55 están las funciones restantes de la clase ColaSec.

23/3/98	vacíaCola (): Lógico	
{ pre: inicio $\geq 0 \wedge$ fin $\geq 0 \wedge$ max = 100}		{pos: inicio $\geq 0 \wedge$ fin $\geq 0 \wedge$ max =100 }
1	regrese (inicio = $0 \wedge$ fin = 0)	**-inicio.** Definido en ColaSec.
1	cola1 = (0, 0, 100) $\Rightarrow$ Verdadero	La variable cola1 esta vacía.

Figura 1.53. Verifica si la cola secuencial está vacía. O(1).

23/3/98	numEle():Entero+	
{pre: inicio $\geq 0 \wedge$ fin $\geq 0 \wedge$ max = 100 }		{pos: inicio $\geq 0 \wedge$ fin $\geq 0 \wedge$ max = 100 }
1	Si (inicio = $0 \wedge$ fin = 0) entonces regrese 0 sino regrese (fin – inicio) fsi	**-inicio, fin.** Definidos en ColaSec.
1	cola1 = (0, 0, 100) $\Rightarrow$ 0	La variable cola1 tiene 0 elementos

Figura 1.54. Regresa el número de elementos actuales en la cola secuencial. O(1).

23/3/98	despliegue()	
{pre: inicio $\geq 0 \wedge$ fin $\geq 0 \wedge$ max = 100 }		{pos: inicio $\geq 0 \wedge$ fin $\geq 0 \wedge$ max = 100 }
1 2	Despliegue “Elementos de la cola” Si (inicio $\neq 0 \wedge$ fin $\neq 0$) entonces [Despliegue “Elemento(“,i,”) = “, c(i)] i = inicio, fin - 1 fsi	**-i:** Entero +: Subíndice del lazo.
1	cola1 = (0, 2, 100, 71, 4) $\Rightarrow$ Elementos de la cola | Elemento(0) = 71 | Elemento(1) = 4	Despliega el contenido de la variable cola1 con sus comentarios.

Figura 1.55. Despliega el contenido de la cola secuencial. O(n).

El orden de complejidad de la clase viene dado por la función que domina la complejidad de todas las funciones de la clase, en este caso O(n).

Cola enlazada:

Una fila o cola implementada con el método enlazado hace uso de otra clase o estructura que define como es el nodo de la cola, en este caso se usa la misma clase Nodo[TipoEle] definida para la pila en la figura 1.38. Esta implementación se puede observar en la figura 1.56 que define la estructura de las variables declaradas con este tipo. Como el método no utiliza localidades de memoria contigua, pues sus nodos están dispersos en la misma, se debe prever un campo para llevar el control del número de elementos actuales en la cola, además de los indicadores del inicio y del fin de la misma. En este caso, ellos son del tipo **ApuntadorA Nodo[TipoEle]** por estar utilizando nodos dispersos en la memoria.

La condición de cola vacía varia por el método de implementación y será inicio = fin = Nulo, para indicar que no hay ningún nodo en la cola. El constructor de la figura 1.57 se encarga de crear e iniciar la variable declarada de este tipo, colocando los valores correspondientes en los campos de su estructura. El destructor presentado en la figura 1.58 se encarga de devolver todos los nodos usados en la cola antes de destruir la variable tipo ColaEnl. Por tener que hacer el barrido de la misma tiene un orden de complejidad lineal.

ColaEnl[TipoEle] Clases: Entero, Lógico, ApuntadorA, TipoEle, Nodo[TipoEle]		
1	**Superclases:** ninguna	**-pInicio**: Dirección del nodo al inicio de la cola.
2	**Estructura:** privado pInicio: ApuntadorA Nodo[TipoEle] = Nulo pFin: ApuntadorA Nodo[TipoEle] = Nulo n: Entero+ = 0	**-pFin**: Dirección del nodo al final de la cola. **-n:** Número de elementos en la cola. **-ColaEnl()**. *Constructor.* Crea una cola vacía. **-~ColaEnl():** *Destructor.* Destruye la cola devolviendo todos sus nodos.
3	**Operaciones:** público ColaEnl() ~ColaEnl() entrarCola(TipoEle: e) salirCola() conCola(): TipoEle vaciaCola(): Lógico numEle(): Entero+ despliegue()	**-entrarCola()**. *Transformador.* Ingresa un nuevo elemento al final de la cola. **-salirCola()**.*Transformador.* Elimina el elemento que se encuentra en el inicio de la cola, si la cola no está vacía. **-conCola()**. *Observador.* Devuelve el elemento que se encuentra actualmente en el inicio de la cola, si existe. **-vacíaCola()**. *Observador.* Regresa verdadero si la cola está vacía. **-numEle()**. *Observador.* Regresa el número actual de elementos, si no está vacía. **-despliegue()**. *Observador.* Despliega el contenido de la cola.

Figura 1.56. Implementación de la clase ColaEnl[TipoEle] según el método enlazado.

23/3/98 ColaEnl() {pos: pInicio = Nulo ∧ pFin = Nulo ∧ n = 0 }		
1	pInicio, pFin, max = Nulo, Nulo, 0	**-pInicio, pFin, n:** Definidos en Cola.
1	ColaEnl[Entero] cola1 ⇒ cola1.n = 0, cola1.pInicio = Nulo, cola1.pFin = Nulo	Crea la variable cola1 de enteros.

Figura 1.57. Constructor de cola enlazada vacía. O(1).

Para ingresar un nuevo elemento en la cola se debe primero verificar si hay memoria para el nodo nuevo que lo contendrá. Ello se realiza solicitando la memoria disponible del tipo Nodo y luego averiguando si esa dirección es diferente del valor nulo. En caso de tenerla, entonces se debe llenar el nodo primero y luego se inserta en la cola, siempre al final de la

misma. En caso que sea la primera inserción, se debe actualizar el valor de la dirección en el campo **pInicio**. El resto de las inserciones se hace de la misma forma. El algoritmo correspondiente a esta función se puede observar en la figura 1.59.

23/3/98 ~ColaEnl()		
{pre: pInicio ≠ Nulo ∧ pFin ≠ Nulo ∧ n ≥ 0 }		
1	(pInicio ≠ Nulo) [pAux = pInicio pInicio = pInicio → PSig() devuelvaDir pAux]	**-pInicio.** Definido en Cola. **-pAux:** ApuntadorA Nodo[TipoEle]. Variable auxiliar.

Figura 1.58. Destructor de la clase ColaEnl[TipoEle]. O(n).

La siguiente figura, la 1.60, contiene el algoritmo de la rutina de eliminación. Ella debe prever los casos de que la cola esté vacía, de que sea la eliminación del último nodo y de que sea una eliminación diferente de las anteriores. En caso que sea el último nodo, deben recolocarse los campos de **pInicio** y **pFin** en Nulo.

23/3/98 entrarCola(TipoEle: e)		
{pre: e ≠ {TipoEleNoDef} ∧ n ≥ 0 }		{ pos: n ≥ 0 }
1 2	pAux = nuevaDir Nodo Si (pAux ≠ Nulo) entonces pAux → Info(e) Si (n = 0) entonces pInicio = pAux sino pFin → PSig(pAux) fsi pFin = pAux n = n + 1 sino Despliegue “Memoria llena” fsi	**-e:** TipoEle. Nuevo elemento a ser insertado **-pInicio, pFin, n:** Definidos en Cola. **-pAux:** ApuntadorA Nodo[TipoEle]. Variable auxiliar para almacenar el nodo nuevo. **-Info(), PSig():** Definidos en Nodo.
1 2	cola1=(0, /, /) , e = 3 ⇒ cola1=(1, →, →) (3, /) cola1=(1, →, →) (3, /), e = 8 ⇒ Memoria llena	Inserta 3 en la cola No hay memoria disponible

Figura 1.59. Ingreso de un nuevo elemento a la cola enlazada. O(1).

La función de consulta de la figura 1.60 regresa una copia del valor almacenado en el nodo que está al inicio de la cola, si ella no está vacía. En vista de que sólo hace una decisión sencilla, el orden de complejidad de la función es constante O(1).

23/3/98 salirCola()		
{pre: n ≥ 0 }		{pos: n ≥ 0}
1	Si (n = 0) entonces Despliegue "Cola vacía" sino pAux = pInicio pInicio = Si (pInicio = pFin) entonces pFin = Nulo Nulo sino pInicio → PSig() fsi devuelvaDir pAux n = n - 1 fsi	**-pInicio, pFin, n:** Definidos en Cola. **-pAux:** ApuntadorA Nodo[TipoEle]. Variable auxiliar para almacenar el nodo nuevo. **-Info(), PSig():** Definidos en Nodo.
1	cola1 = (1, →, →) (3, /)⇒ cola1 = (0, /, /)	Elimina el último elemento
2	cola1 = (0, /, /) ⇒ Cola vacía	No elimina, cola vacía

Figura 1.60. Eliminación del elemento al inicio de la cola enlazada. O(1).

23/3/98 conCola():TipoEle		
{pre: n ≥ 0 }		{pos: n ≥ 0 }
1	Si (n = 0) entonces regrese {TipoEleNoDef} sino regrese (pInicio → Info()) fsi	**-n, pInicio.** Definidos en la clase Cola. **-Info(), {TipoEleNoDef}.** Definidos en la clase Nodo.
1	cola1 = (1, →, →) (3, /) ⇒ 3	Se obtiene una copia del elemento en el inicio
2	cola2 = (0, /, /) ⇒ -200	Se obtiene una copia del valor no definido

Figura 1.61. Consulta del elemento al inicio de la cola enlazada. O(1).

23/3/98 vacíaCola(): Lógico		
{ pre: n ≥ 0 }		{pos: n ≥ 0 }
1	regrese (n = 0)	**-n.** Definido en la clase Cola.
1	cola1 = (0, /, /) ⇒ Verdadero	La variable cola1 está vacía.
2	cola1 = (1, →, →) (3, /) ⇒ Falso	No está vacía la variable cola1.

Figura 1.62. Verificación de cola vacía. O(1).

23/3/98 numEle():Entero+		
{pre: n ≥ 0 }		{pos: n ≥ 0 }
1	regrese n	**-n:** Definido en la clase Cola .
1	cola1 = (0, /, /) ⇒ 0	La variable cola1 tiene 0 elementos.

Figura 1.63. Regresa el número actual de elementos en la cola enlazada. O(1).

El resto de las funciones presentadas en las figuras de la 1.61 a la 1.64 realizan la verificación de cola vacía, la observación de cuántos elementos hay actualmente en la cola y el despliegue de los elementos en la cola. Esta última es la que tiene una complejidad lineal,

ya que debe recorrer la cola para poder hacer el despliegue solicitado. La complejidad total de la clase ColaEnl[TipoEle] es entonces lineal O(n).

23/3/98 despliegue() {pre: n ≥ 0 }		{pos: n ≥ 0 }
1	Si (n = 0) entonces Despliegue "La cola está vacía" sino Despligue "Elementos de la cola" pAux, m = pInicio, 1 (pAux ≠ Nulo)[Despliegue "Elemento(", m, ") = ", pAux → Info() pAux, m = pAux → PSig(), m + 1] fsi	**-m:** Entero +: Subíndice del lazo. **-pAux:** ApuntadorA Nodo[TipoEle]. Variable auxiliar para recorrer la cola. **-Info(), PSig():** Definidos en la clase Nodo.
1	cola1 = (0, /, /) ⇒ La cola está vacía	Despliega el comentario de cola vacía.
2	cola1= (2, →, →2do.nodo) (3, →) (8, /) ⇒ Elementos de la cola | Elemento(1) = 3 | Elemento(2) = 8	Despliega el contenido de cola1 con sus comentarios.

Figura 1.64. Despliega el contenido de la cola enlazada. O(n).

Cola secuencial circular:

Para evitar el problema de espacio no utilizado en la cola secuencial, se presenta ésta implementación, donde el arreglo que sirve para contener los elementos en la cola se dobla para formar un anillo uniendo la primera con la última posición del arreglo. La ilustración se puede observar en la figura 1.65. En este caso, ambos indicadores de la cola son incrementados como es habitual, pero al llegar cualquiera de los dos a la máxima posición, ellos son reiniciados a 0 si la localidad está disponible.

La figura 1.66 muestra la implementación de la clase ColaCir[TipoEle] para la cola circular secuencial. Para ella la condición de cola vacía cambia a inicio = fin = -1. Por lo que el constructor de la figura 1.67 iniciará los campos de la estructura de la cola en dichos valores.

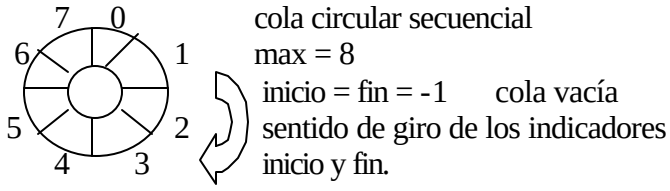


Figura 1.65. Cola secuencial circular.

Conforme a la condición de cola vacía del constructor, la rutina de inserción debe actualizar los indicadores en cada llamada, por lo que debe detectar los casos de: primera inserción o inserción en cola vacía, ingreso del nuevo elemento cuando el fin ya está en el máximo para recolocarlo en cero y el resto de las inserciones que se realizan de la misma forma. La figura 1.68 presenta el algoritmo correspondiente para la función de ingresar un nuevo elemento a la cola.

ColaCir[TipoEle] Clases: Entero, Lógico, TipoEle, ArregloDe		
1	**Superclases:** ninguna	**-inicio**: Subíndice del primer elemento.
2	**Estructura:** privado	**-fin**: Subíndice del último elemento.
	inicio: Entero+ = -1	**-max:** Número máximo de elementos
	fin: Entero+ = -1	**-c**: Arreglo que contiene los elementos de la cola.
	max: Entero+ = 100	**-ColaCir()**. *Constructor.* Crea una cola vacía.
	c : Arreglo[100]De TipoEle	**-entrarCola()**. *Transformador.* Ingresa un nuevo elemento por el fin de la cola.
3	**Operaciones:** público	**-salirCola()**.*Transformador.* Elimina el elemento que se encuentra en el inicio de la cola, si la cola no esta vacía.
	ColaCir():	**-conCola()**. *Observador.* Devuelve el elemento que se encuentra actualmente en el inicio de la cola, si existe
	entrarCola(TipoEle: e)	**-vacíaCola()**. *Observador.* Regresa verdadero si la cola está vacía.
	salirCola()	**-numEle()**. *Observador.* Regresa el número actual de elementos, si no esta vacía.
	conCola(): TipoEle	**-despliegue()**. *Observador.* Despliega el contenido actual de la cola.
	vaciaCola(): Lógico	
	numEle(): Entero+	
	despliegue()	

Figura 1.66. Implementación de la clase cola circular secuencial.

23/3/98 ColaCir() {pos: inicio = -1 ∧ fin = -1 ∧ max =100}		
1	inicio, fin, max = -1, -1, 100	**-inicio, fin, max:** Definidos en ColaCir.
1	ColaCir[Entero] cola1 ⇒ cola1.max = 100, cola1.inicio = -1, cola1.fin = -1	Crea la variable cola1 de enteros.

Figura 1.67. Constructor vacío. O(1).

23/3/98 entrarCola(TipoEle: e) {pre: e ≠ {TipoEleNoDef} ∧ -1 ≤ inicio, fin < max} { pos: C´= C + {e} ∧ 0 ≤ inicio, fin < max }		
1	fin = fin + 1	**-e:** TipoEle. Nuevo elemento a ser insertado.
2	Si (fin = max) entonces fin = 0 fsi	**-inicio, fin, max:** Definidos en ColaCir.
3	Si (fin = inicio) entonces Despliegue "Cola llena" sino c (fin) = e Si (inicio = -1) entonces inicio = 0 fsi fsi	
1	cola1=(5,99,100,71,4),e = 3 ⇒ cola1=(5,0,100,71,4,3)	Inserta 3 en la cola.
2	cola1=(0,99,100,7,..), e =8 ⇒ cola1= (0,99,100,7,..)	No hace la inserción, la cola está llena.

Figura 1.68. Inserción de un nuevo elemento en la cola secuencial circular. O(1).

Las mismas consideraciones se hacen para la rutina de eliminación, esas son: eliminación del último elemento, supresión de un elemento en el máximo, por lo que hay que reposicionar el inicio en cero y el resto de las supresiones que se realizan con las mismas instrucciones. La función se puede observar en la figura 1.69.

23/3/98 salirCola()		
	{ pre: -1 ≤ inicio, fin < max }	{pos: -1 ≤ inicio, fin < max }

1	Si (fin = -1) entonces
	  Despliegue "Cola vacía"
	sino
	  Si (inicio = fin) entonces
	  inicio = fin = -1
	  sino
	  inicio = inicio + 1
	  Si (inicio = max) entonces
	  inicio = 0
	  fsi
	  fsi
	fsi
		-inicio, fin, max: Definidos en ColaCir.

1 2 3 4	cola1= (0, 2, 100, 71, 4) ⇒ cola1= (1, 2, 100, 71, 4) cola1= (99,3,100, 71, 4, 32) ⇒ cola1(0,3,100,71,4,32) cola1= (-1, -1, 100) ⇒ cola1(-1, -1, 100) cola1= (3, 3, 100, 23) ⇒ cola1(-1, -1, 100)	Elimina el elemento en inicio Elimina el elemento en inicio No elimina, cola vacía Elimina el único elemento de la cola

Figura 1.69. Eliminación del elemento al inicio de la cola secuencial circular. O(1).

El resto de las funciones mostradas en las figuras de la 1.70 a la 1.73 se realizan parecidas a las rutinas de las colas secuenciales, pero teniendo la condición de cola vacía cambiada por la de la cola circular.

23/3/98 ConCola():TipoEle		
{pre: -1 ≤ inicio < max}		{pos: -1 ≤ inicio < max }

1	Si (inicio = -1) entonces
	  regrese {TipoEleNoDef}
	sino
	  regrese c (inicio)
	fsi
		-inicio, fin, c. Definidos en ColaCir.

1 2	cola1 = (0, 1, 100, 71, 4) ⇒ 71 cola1 = (-1, -1, 100) ⇒ -200	Se obtiene una copia del elemento en el inicio Se obtiene el elemento especial

Figura 1.70. Consulta del elemento al inicio de la cola secuencial circular. O(1).

23/3/98 vacíaCola (): Lógico		
{ pre: -1 ≤ inicio, fin < max ∧ max = 100}		{pos: -1 ≤ inicio, fin < max ∧ max =100 }
1	regrese (inicio = -1 ∧ fin = -1)	**-inicio, fin.** Definido en ColaCir.
1	cola1 = (-1, -1, 100) ⇒ Verdadero	La variable cola1 esta vacía.

Figura 1.71. Verifica si la cola secuencial circular está vacía. O(1).

23/3/98	numEle():Entero+	
{pre: -1 ≤ inicio, fin < max ∧ max = 100 }		{pos: -1 ≤ inicio, fin < max ∧ max = 100 }

1	Si (inicio = -1 ∧ fin = -1) entonces
	  regrese 0
	sino Si (fin ≥ inicio) entonces
	  regrese (fin – inicio + 1)
	  sino
	  regrese (max – inicio + fin + 1)
	  fsi
	fsi
		-inicio, fin. Definidos en ColaCir.

1 2 3	cola1 = (-1, -1, 100) ⇒ 0 cola1 = (2, 6, 100, 5, 70,...) ⇒ 5 cola1 = (12, 6, 100, 5, 70,...) ⇒ 95	La variable cola1 tiene 0 elementos. Hay cinco elementos en cola1. Hay 95 elementos en cola1.

Figura 1.72. Regresa el número de elementos actuales en la cola secuencial circular. O(1).

23/3/98	despliegue()	
{pre: -1 ≤ inicio, fin < max ∧ max = 100 }		{pos: -1 ≤ inicio, fin < max ∧ max = 100 }

1
2
	Si (inicio = -1) entonces
	  Despliegue “Cola vacía”
	sino
	  Despliegue “Elementos de la cola”
	  Si (inicio < fin) entonces
	  [Despliegue “Elemento(“,i,”) = “, c(i)] i = inicio, fin
	  sino
	  [Despliegue “Elemento(“,i,”) = “, c(i)] i =inicio, max-1
	  [Despliegue “Elemento(“,i,”) = “, c(i)] i = 0, fin
	  fsi
	fsi
		-i: Entero +: Subíndice del lazo.

1	cola1 = (0, 2, 100, 71, 4) ⇒ Elementos de la cola | Elemento(0) = 71 | Elemento(1) = 4	Despliega el contenido de la variable cola1 con sus comentarios.

Figura 1.73. Despliega el contenido de la cola secuencial circular. O(n).

La complejidad de la clase ColaCir[TipoEle] será lineal debido a la rutina de despliegue.

1.18.3. TAD: Dipolo según los métodos secuencial y enlazado simple

Dipolo secuencial:

Esta estructura lineal de datos tiene poco uso en aplicaciones reales, sin embargo se incluyen aquí las dos implementaciones básicas de la misma. La figura 1.74 presenta la implementación de un dipolo secuencial que al igual que los anteriores, usa la asignación de memoria contigua con un tamaño fijo. La implementación de un dipolo enlazado simple se puede observar en la figura 1.75.

La implementación de cada rutina o función especificada en las implantaciones de las figuras 1.74 y 1.75 quedan de ejercicio para el lector.

DipoloSec[TipoEle] Clases: Entero, Lógico, TipoEle, ArregloDe		
1	**Superclases:** ninguna	**-max**: Número máximo de elementos.
2	**Estructura**: privado: max : Entero+ = 100 izq : Entero+ = 0 der : Entero+ = 0 d : Arreglo[100]De TipoEle	**-izq**: Indicador del elemento más a la izquierda. **-der:** Indicador del elemento más a la derecha. **-d:** Arreglo que contiene los elementos actuales en el dipolo. **-DipoloSec()**. *Constructor.* Crea un dipolo vacío. **-insIzqDipolo(), insDerDipolo()**. *Transformadores*. Ingresan un nuevo elemento antes del más a la izquierda y luego del más a la derecha del dipolo, respectivamente.
3	**Operaciones:** público: DipoloSec() insIzqDipolo(TipoEle: e) insDerDipolo(TipoEle: e) eliIzqDipolo() eliDerDipolo() conIzqDipolo(): TipoEle conDerDipolo(): TipoEle vacíoDipolo(): Lógico numEle(): Entero+ despliegue()	**-eliIzqDipolo(), eliDerDipolo()**. *Transformadores*. Eliminan el elemento más a la izquierda y el más a la derecha del dipolo, respectivamente, si no está vacío. **-conIzqDipolo(), conDerDipolo()**. *Observadores*. Devuelven el elemento más a la izquierda y el más a la derecha del dipolo, respectivamente, si existen. **-vacíoDipolo()**. *Observador.* Regresa Verdadero si el dipolo está vacío. **-numEle()**. *Observador.* Regresa el número actual de elementos en el dipolo. **-despliegue().** *Observador.* Despliega el contenido del dipolo

Figura 1.74. Implementación según el método secuencial.

DipoloEnl[TipoEle] Clases: Entero, Lógico, ApuntadorA, TipoEle, Nodo		
1	**Superclases:** ninguna	**-n:** Entero+. Número actual de elementos en el dipolo.
2	**Estructura**: privado: n: Entero+ = 0 izq : ApuntadorA Nodo[TipoEle] = Nulo der : ApuntadorA Nodo[TipoEle] = Nulo	**-izq**: ApuntadorA Nodo. Apuntador al nodo más a la izquierda. **-der:** ApuntadorA Nodo. Apuntador al nodo más a la derecha. **-DipoloEnl()**. *Constructor.* Crea un dipolo vacío. **-~DipoloEnl():** *Destructor.* Destruye el dipolo.
3	**Operaciones:** público: DipoloEnl() ~DipoloEnl() insIzqDipolo(TipoEle: e) insDerDipolo(TipoEle: e) eliIzqDipolo() eliDerDipolo() conIzqDipolo(): TipoEle conDerDipolo(): TipoEle vacíoDipolo(): Lógico numEle(): Entero+ despliegue()	**-insIzqDipolo(), insDerDipolo()**. *Transformadores*. Ingresan un nuevo elemento antes del más a la izquierda y luego del más a la derecha del dipolo, respectivamente. **-eliIzqDipolo(), eliDerDipolo()**. *Transformadores*. Eliminan el elemento más a la izquierda y el más a la derecha del dipolo, respectivamente, si no está vacío. **-conIzqDipolo(), conDerDipolo()**. *Observadores*. Devuelven el elemento más a la izquierda y el más a la derecha del dipolo, respectivamente, si existen. **-vacíoDipolo()**. *Observador.* Regresa Verdadero si el dipolo está vacío. **-numEle()**. *Observador.* Regresa el número actual de elementos en el dipolo. **-despliegue().** *Observador.* Despliega el contenido del dipolo

Figura 1.75. Implementación de un dipolo según el método enlazado simple.

1.18.4. TAD: Lista según los métodos: secuencial, enlazado simple, enlazado doble y enlazado doble circular

Lista secuencial:

Se presenta en la figura 1.76 la implementación de la lista según el método secuencial.

ListaSec [TipoEle] Clases: Entero, Lógico, TipoEle, ArregloDe		
1	**Superclases:** ninguna	-**max**: Número máximo de elementos.
2	**Estructura**: privado: max : Entero+ = 100 n : Entero+ = 0 l : Arreglo[100]De TipoEle	-**n**: Indicador del número actual de elementos. **-l:** Arreglo que contiene los elementos actuales en la lista. -**ListaSec()**. *Constructor.* Crea una lista vacía.
3	**Operaciones:** público: ListaSec() insLista(TipoEle: e, Entero+: p) eliLista(Entero+: p) conLista(Entero+: p): TipoEle vacíaLista(): Lógico numEle(): Entero+ despliegue()	-**insLista()**. *Transformador.* Ingresa un nuevo elemento en la posición p. -**eliLista()**. *Transformador.* Elimina el elemento en la posición p, si la lista no está vacía. -**conLista()**. *Observador.* Devuelve el elemento de la posición p, si existe. -**vacíaLista()**. *Observador.* Regresa Verdadero si la lista está vacía. -**numEle()**. *Observador.* Regresa el número actual de elementos en la lista. -**despliegue().** *Observador.* Despliega el contenido de la lista.

Figura 1.76. Implementación una lista según el método secuencial.

Se considera que las inserciones de nuevos elementos y las eliminaciones y consultas de elementos en la lista se hacen según la posición indicada por la variable **p**. Por ello, **p** debe variar entre uno y el tamaño actual de la lista, si p = 1, la operación se realiza en el primer elemento y si p = tamaño actual, se hace sobre el último elemento de la lista. El constructor de una lista vacía se puede observar en la figura 1.77.

Las figuras 1.78 y 1.79 muestran la inserción y la eliminación, respectivamente. Para insertar un nuevo elemento en una lista secuencial se debe verificar si la lista está llena, en cuyo caso es imposible insertar algo. Si no está llena, entonces se debe verificar en que punto se desea la inserción, al final no se necesita mover ningún elemento, en cualquier otro punto, es necesario mover los elementos para abrir una brecha o espacio, para almacenar al nuevo elemento. Consideraciones parecidas se deben hacer en caso de una eliminación. Primero se verifica si hay algo en la lista. Si no está vacía, entonces se verifica si la eliminación es al final, para lo cual no se mueve ningún elemento, de lo contrario se cierra la brecha o espacio, para que todos los elementos queden seguidos y no hayan huecos intermedios.

27/3/98 ListaSec() {pos: n = 0 $\wedge$ max = 100}		
1	max, n = 100, 0	**-max, n.** Definidos en ListaSec.
1	ListaSec[Entero] lis $\Rightarrow$ lis.max = 100 y lis.n = 0	Crea la variable lis de enteros.

Figura 1.77. Constructor vacío. O(1).

27/3/98		
	insLista(TipoEle: e, Entero+: p)	
{pre: e ≠ {TipoEleNoDef} ∧ 0 ≤ n < max ∧ 0 < p ≤ n+1 }		{pos: 0 ≤ n < max }

1	Si (n = max) entonces
	  Despliegue "Lista llena"
	  sino Si (p ≥ n) entonces
	  l (n) = e
	  sino
	  [l (m) = l (m - 1)] m = n, p, -1
	  l (p - 1) = e
	  fsi
	  n = n + 1
	fsi
		-e: TipoEle. Nuevo elemento para ser insertado en la lista.
		-n, max, l, p: Definidos en ListaSec.
		-m: Entero+. Subíndice del lazo de recorrido de la lista.

1	lis=(100, 2, 45, 34), e = 8, p = 3 ⇒ lis=(100, 3, 45, 34, 8)	Inserta 8 al final.
2	lis=(100,100, 45, ...), e=8, p=3 ⇒ lis=(100,100, 45, ...)/Lista llena	No inserta 8 pues está llena.
3	lis=(100, 2, 45, 34), e = 8, p = 1 ⇒ lis=(100, 3, 8, 45, 34)	Inserta 8 al inicio.
4	lis=(100, 2, 45, 34), e = 8, p = 2 ⇒ lis=(100, 3, 45, 8, 34)	Inserta 8 en el medio.

Figura 1.78. Ingresa un nuevo elemento en la lista. O(n).

27/3/98		
	eliLista(Entero+: p)	
{pre: 0 ≤ n < max ∧ 0 < p ≤ n }		{pos: 0 ≤ n < max }

1	Si (n = 0) entonces
	  Despliegue "Lista vacía"
	  sino Si (p > n) entonces
	  Despliegue "Ese elemento no está en la lista"
	  sino
	  [l (m - 1) = l (m)] m = p, (n - 1)
	  n = n - 1
	  fsi
	fsi
		-m: Entero+. Subíndice de recorrido de la lista.
		- n, l, p: Definidos en ListaSec.

1	lis=(100, 2, 45, 34), p = 2 ⇒ lis=(100, 1, 45)	Elimina al final.
2	lis=(100,0), p=3 ⇒ lis=(100,0) / Lista vacía	No elimina pues está vacía.
3	lis=(100, 2, 45, 34), p = 1 ⇒ lis=(100, 1, 34)	Elimina al inicio.
4	lis=(100, 3, 45, 34, 8), p = 2 ⇒ lis=(100, 2, 45, 8)	Elimina en el medio.
5	lis=(100, 2, 45, 34), p = 5 ⇒ lis=(100, 2, 45, 34) / Ese elemento no está en la lista	No elimina pues la posición solicitada está fuera de rango

Figura 1.79. Elimina un elemento de la lista. O(n).

27/3/98		
	conLista(Entero+: p): TipoEle	
{pre: 0 ≤ n < max ∧ 0 < p ≤ n }		{pos: 0 ≤ n < max }

1	Si (n = 0 ∨ p > n) entonces
	  regrese {TipoEleNoDef}
	  sino
	  regrese l (p - 1)
	fsi
		- n, l, p: Definidos en ListaSec.

1	lis=(100, 2, 45, 34), p = 2 ⇒ 34	Regresa el segundo elemento
2	lis=(100,0), p=3 ⇒ -200	Regresa un no definido pues está vacía.

Figura 1.80. Consulta un elemento de la lista. O(1).

Las figuras desde la 1.80 a la 1.83 presenta el resto de las funciones de la clase ListaSec[TipoEle], por lo cual se deduce que el orden de complejidad de la clase es O(n). Todos los elementos de la lista están uno detrás del otro **sin** huecos intermedios.

27/3/98 vacíaLista(): Lógico		
{ pre: 0 ≤ n < max }		{pos: 0 ≤ n < max }
1	regrese (n = 0)	**-n.** Definido en ListaSec.
1	lis = (100, 0) ⇒ Verdadero	Lista vacía.
2	lis =(100, 2, 45, 34) ⇒ Falso	Lista no vacía.

Figura 1.81. Verifica si la lista está vacía. O(1).

27/3/98 numEle (): Entero+		
{ pre: 0 ≤ n < max }		{pos: 0 ≤ n < max }
1	regrese n	**-max, n.** Definidos en ListaSec.
1	lis = (100, 0) ⇒ 0	No hay elementos en la lista.
2	lis=(100, 3, 45, 34, 8) ⇒ 3	Hay 3 elementos en la lista.

Figura 1.82. Devuelve el número actual de elementos en la lista. O(1).

27/3/98 despliegue()		
{ pre: 0 ≤ n < max }		{pos: 0 ≤ n < max }

1	Si (n = 0) entonces
	  Despliegue “No hay elementos en la lista”
	sino
	  Despliegue “Elementos de la lista”
	  [Despliegue “e (“, m+1, “)=”, l (m)] m=0, n–1
	fsi
		-max, n, l. Definidos en ListaSec.
		-m: Entero+. Subíndice de recorrido de la lista.

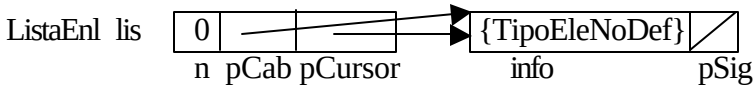
1	lis=(100, 3, 45, 34, 8) ⇒ Elementos de la lista / e (1)= 45 / e (2)= 34 / e (3)= 8	Despliega el contenido actual de lis.
2	lis = (100, 0) ⇒ No hay elementos en la lista	Lista vacía.

Figura 1.83. Despliega en pantalla el contenido de la lista. O(n).

Lista enlazada simple:

Para esta implementación se considera la inclusión de un nodo especial en la lista denominado **cabeza**, que sirve para mejorar los algoritmos de las funciones en lo que respecta a su número de instrucciones y que además permite efectuar las operaciones de inserción, eliminación y consulta más rápidamente. Con el nodo especial, se define una variable adicional denominada **cursor** que apunta al nodo actual en la lista y se especifican las operaciones antes mencionadas, para que se efectúen **después** del nodo actual. Cuando **cursor** apunta a **cabeza** se puede hacer una inserción antes del primer nodo y la eliminación y consulta del primer nodo de la lista. La figura 1.84 ilustra la implementación de la clase ListaEnl con estas características.

Para esta clase la condición de lista vacía podrá ser n = 0 o bien pCab → PSig() = Nulo, ya que al crear la ListaEnl[TipoEle] se pide el nodo especial, que será llenado con el valor no definido del TipoEle y el valor Nulo para su apuntador. Gráficamente se tiene:



En este caso el constructor de la clase debe solicitar el nodo que servirá de cabeza de la lista, lo que se observa en la figura 1.85.

ListaEnl[TipoEle] Clases: Entero, Lógico, ApuntadorA, TipoEle, Nodo[TipoEle]		
1	**Superclases:** ninguna	**-n**: Número actual de elementos.
2	**Estructura**: privado: n : Entero+ = 0 pCab : ApuntadorA Nodo[TipoEle] = Nulo pCursor : ApuntadorA Nodo[TipoEle] = Nulo	**-pCab**: Apuntador al nodo especial. **-pCursor:** Apuntador al nodo actual. **-ListaEnl()**. *Constructor.* Crea una lista vacía. **-~ListaEnl().** *Destructor.* Destruye la lista. **-insLista()**. *Transformador.* Ingresa un nuevo elemento luego del nodo actual.
3	**Operaciones:** público: ListaEnl() ~ListaEnl() insLista(TipoEle: e) eliLista() conLista(): TipoEle vacíaLista(): Lógico numEle(): Entero+ cursorAlInicio() cursorAlProximo(): Lógico despliegue()	**-eliLista()**. *Transformador.* Elimina el elemento después del nodo actual, si la lista no está vacía. **-conLista()**. *Observador.* Devuelve el elemento después del nodo actual, si existe. **-vacíaLista()**. *Observador.* Regresa Verdadero si la lista está vacía. **-numEle()**. *Observador.* Regresa el número actual de elementos en la lista. **-cursorAlInicio()**. *Transformador.* Coloca el cursor antes del primer nodo. **-cursorAlProximo().** *Transformador.* Avanza el cursor al próximo nodo, si fue posible regresa Verdadero. **-despliegue().** *Observador.* Despliega el contenido de la lista.

Figura 1.84. Implementación de una lista según el método enlazado simple.

27/3/98 ListaEnl() {pos: n = 0 ∧ pCab → PSig() = Nulo }		

1
2
	n, pCab = 0, nuevaDir Nodo
	Si (pCab = Nulo) entonces
	  Despliegue “Memoria llena”
	sino
	  pCursor = pCab
	fsi
		-n, pCab, pCursor. Definidos en ListaEnl.
		-nuevaDir. Regresa una dirección de memoria disponible del tipo Nodo.

1	ListaEnl lis ⇒ lis.n = 0, lis.pCab ≠ Nulo y lis.pCursor ≠ Nulo	Crea la variable lis de este tipo.

Figura 1.85. Constructor vacío. O(1).

El destructor de esta clase mostrado en la figura 1.86, debe devolver toda la memoria ocupada por los nodos que están asignados a la lista, incluyendo el nodo especial.

27/3/98 ~ListaEnl() {pre: n ≥ 0}		
1	(pCab ≠ Nulo) [pAux = pCab pCab = pCab → PSig() devuelvaDir pAux]	**-pCab.** Definido en ListaEnl. **-PSig():** Definido en la clase Nodo. **-pAux:** ApuntadorA Nodo[TipoEle]. Variable auxiliar

Figura 1.86. Destruye la lista enlazada. O(n).

La función de inserción presentada en la figura 1.87, ilustra la sencillez del algoritmo, ya que las mismas instrucciones son usadas para los tres casos de inserción en listas, estos son: inserción en vacío o al inicio, inserción en el medio e inserción al final.

27/3/98	insLista(TipoEle: e)	
{pre: e ≠ {TipoEleNoDef} ∧ n ≥ 0}		{pos: n ≥ 0}
1 2	pAux = nuevaDir Nodo Si (pAux ≠ Nulo) entonces pAux → Info(e) pAux → PSig(pCursor → PSig()) pCursor → PSig(pAux) n = n + 1 sino Despliegue "Memoria llena" fsi	**-pAux:** ApuntadorA Nodo[TipoEle]. Variable auxiliar. **-Info(), PSig():** Definidos en la clase Nodo. **-n, pCursor**: Definidos en ListaEnl. **-nuevaDir Nodo:** Regresa una dirección de memoria disponible de tipo Nodo[TipoEle].
1	lis = (0, →, →) (-8, /), e = 78 ⇒ lis = (1, →, →) (-8, →) (78, /)	Ingresa 78 en una lista vacía. El elemento no definido es –8.
2	lis = (2, →, →) (-8, →) (45, →) (34, /), e = 20 ⇒ lis = (3, →, →) (-8, →) (20, →) (45, →) (34, /)	Ingresa 20 al inicio de la lista, es decir después del nodo cabeza.

Figura 1.87. Inserta un nuevo elemento en la lista. O(1).

La figura 1.88 presenta el algoritmo de eliminación que también tiene una complejidad constante debido al uso del nodo especial.

27/3/98	eliLista ()	
{pre: n ≥ 0}		{pos: n ≥ 0}
1	Si (pCab → PSig() ≠ Nulo) entonces pAux = pCursor → PSig() pCursor → PSig(pCursor → PSig( ) →PSig()) devuelvaDir pAux n = n - 1 sino Despliegue "Lista vacía" fsi	**-pAux:** ApuntadorA Nodo[TipoEle]. Variable auxiliar. **-PSig():** Definido en la clase Nodo. **-n, pCursor**: Definidos en ListaEnl. **-devuelvaDir:** Regresa la dirección a la memoria disponible.
1	lis=(0, →, →)(-8, /) ⇒ Lista vacía	No elimina
2	lis = (2, →, →)(-8, →) (45, →) (34, /) ⇒ lis = (1, →, →)(-8, →) (34, /)	Elimina 45 de la lista, siendo –8 el elemento no definido del TipoEle.

Figura 1.88. Elimina un elemento en la lista. O(1).

Igual que en la inserción, el bloque de instrucciones necesario para efectuar la eliminación de un nodo es el mismo para los cuatro casos de eliminación, a saber: eliminación del primer nodo, de un nodo intermedio, del último nodo y del único nodo que queda en la lista.

Las mismas consideraciones anteriores se hacen para el caso de una consulta en la lista enlazada simple de la figura 1.89. Dado el uso del nodo especial los tres casos de consulta: al nodo inicial, a un nodo intermedio y al nodo final, se realizan con la misma instrucción.

27/3/98 conLista(): TipoEle		
{pre: n ≥ 0}		{pos: n ≥ 0}

1	Si (pCab → PSig() = Nulo) entonces
	  regrese pCab → Info()
	sino
	  regrese pCursor → PSig() → Info()
	fsi
		-Info(), PSig(): Definidos en la clase Nodo.
		-n, pCab, pCursor: Definidos en ListaEnl.

1	lis = (0, →, →)(-8, /) ⇒ -8	Devuelve el valor {TipoEleNoDef} = -8
2	lis = (2, →, →)(-8, →) (45, →) (34, /) ⇒ 45	Devuelve 45

Figura 1.89. Devuelve el contenido del nodo de la lista luego del cursor. O(1).

27/3/98 vacíaLista(): Lógico		
{pre: n ≥ 0}		{pos: n ≥ 0}
1	Regresa (n = 0)	**-n.** Definido en ListaEnl.
1	lis = (2, →, →)(-8, →)(34, →)(9, /) ⇒ Falso	No está vacía.
2	lis = (0, →, →)(-8, /) ⇒ Verdadero	La variable lis está vacía.

Figura 1.90. Verifica si la lista está vacía. O(1).

27/3/98 numEle(): Entero+		
{pre: n ≥ 0}		{pos: n ≥ 0}
1	Regresa n	**-n.** Definido en ListaEnl.
1	lis = (2, →, →)(-8, →)(32, →)(88, /) ⇒ 2	Hay dos nodos en la lista.
2	lis = (0, →, →)(-8, /) ⇒ 0	La variable lis tiene 0 elementos.

Figura 1.91. Devuelve el número de elementos actuales en la lista. O(1).

27/3/98 cursorAlInicio()		
{pre: n ≥ 0}		{pos: n ≥ 0 ∧ pCursor = pCab }
1	pCursor = pCab	**-pCursor, pCab.** Definidos en ListaEnl.
1	lis = (2, →, →2do. nodo)(-8, →)(44, →)(88, /) ⇒ lis = (2, →, →)(-8, →)(44, →)(88, /)	Cursor al inicio de la lista.
2	lis = (0, →, →)(-8, /) ⇒ lis = (0, →, →)(-8, /)	Cursor al inicio de la lista.

Figura 1.92. Mueve el cursor al nodo inicial en la lista. O(1).

27/3/98 cursorAlProximo(): Lógico		
{pre: n ≥ 0}		{pos: n ≥ 0}
1 2 3	ban = Falso pCursor, ban = Si (pCursor → PSig() ≠ Nulo) entonces pCursor → PSig() , Verdadero fsi regrese ban	**-pCursor.** Definido en ListaEnl. **-PSig().** Definido en Nodo. **-ban:** Lógico. Variable auxiliar para indicar si pudo o no moverse al próximo nodo.
1	lis = (2, →, →1er. nodo)(-8, →)(56, →)(88, /) ⇒ lis = (2, →, →2do.nodo)(-8, →)(56, →)(88, /)| Verdadero	Coloca el cursor apuntando al segundo nodo de la lista.
2	lis = (0, →, →)(-8, /) ⇒ Falso	No hay próximo nodo.

Figura 1.93. Mueve el cursor al próximo nodo en la lista. O(1).

El resto de las funciones de la clase ListaEnl[TipoEle] se pueden observar en las figuras desde la 1.90 hasta la 1.94, donde todas ellas tienen una complejidad constante a excepción de la función **despliegue()** cuya complejidad depende del tamaño de la lista.

27/3/98 despliegue () {pre: n ≥ 0}		{pos: n ≥ 0}

1	Si (n = 0) entonces
	  Despliegue "La lista está vacía"
	sino
	  Despliegue "Elementos de la lista"
	  pCursor = pCab → Psig()
	  [Despliegue "Elemento(", i , ")=",pCursor → Info()
	  pCursor = pCursor → PSig()] i = 1, n
	fsi
		-pCursor, pCab. Definidos en la clase ListaEnl.
		-Info(), PSig(): Definidos en la clase Nodo.
		-i: Entero+. Subíndice del lazo.

1	lis = (0, →, →)(-8, /) ⇒ La lista está vacía	No hay elementos en la lista.
2	lis = (2, →, →)(-8, →) (45, →) (34, /) ⇒ Elementos de la lista | Elemento(1)=45 | Elemento(2)=34 |	Despliega el contenido de la lista con sus comentarios

Figura 1.94. Despliega los elementos actuales en la lista. O(n).

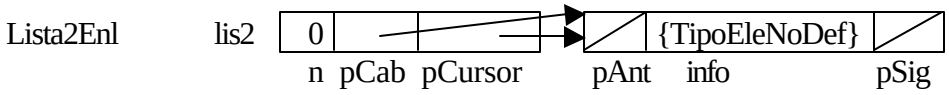
En conclusión, la complejidad de la clase ListaEnl[TipoEle] es lineal O(n).

Lista doblemente enlazada:

Una lista doblemente enlazada contiene unos nodos con dos campos apuntadores, uno que apunta al nodo anterior (pAnt) y el otro que apunta al nodo siguiente (pSig), además del campo que contiene la información que almacena el nodo (info). Este nodo se especifica parecido al nodo con un solo apuntador. La figura 1.95 muestra la implementación, cuyos métodos tiene una complejidad constante O(1), pues cada uno de los métodos que conforman la clase son simples observadores y transformadores para los campos de la clase.

En una implementación en el lenguaje C++, esta clase podría ser incluida dentro de la clase de la lista doblemente enlazada, como una estructura privada de la clase y de esta manera no puede ser vista por ninguna otra clase que utilice la lista.

La implementación de la clase lista doblemente enlazada (Lista2Enl[TipoEle]) se presenta en la figura 1.96 la cual hace uso de la clase Nodo2[TipoEle]. La condición de lista vacía en este caso se ilustra como:



Al igual que para las listas simplemente enlazadas, se usa el nodo especial llamado cabeza de la lista, este nodo siempre tendrá su apuntador al nodo anterior igual a Nulo, pues no habrá nunca un nodo antes que él. Se le anexa una función para mover el cursor al nodo anterior, dada la facilidad de hacerlo al tener un apuntador al nodo anterior.

El constructor mostrado en la figura 1.97 debe solicitar un nodo de dos enlaces vacíos para que pueda ser utilizado como el nodo especial cabeza de la lista. Esos dos enlaces permiten moverse hacia delante y hacia atrás en la lista.

Nodo2[TipoEle] Clases: TipoEle, ApuntadorA		
1	**Superclases:** ninguna	**-pAnt:** Dirección del nodo anterior.
2	**Estructura**: privado: pAnt : ApuntadorA Nodo2[TipoEle] = Nulo info: TipoEle = {TipoEleNoDef} pSig : ApuntadorA Nodo2[TipoEle] = Nulo	-**info**: Información del nodo. -**pSig**: Dirección del nodo siguiente. -**Nodo2()**. *Constructor.* Crea un nodo con un elemento no definido y dos apuntadores nulos. **-PAnt().** *Transformador.* Modifica el apuntador al nodo anterior por la dirección que viene en el parámetro.
3	**Operaciones:** público: Nodo2() PAnt(ApuntadorA Nodo2[TipoEle]: pa) PAnt() : ApuntadorA Nodo2[TipoEle] Info(TipoEle: e) Info(): TipoEle PSig(ApuntadorA Nodo2[TipoEle]: pn) PSig(): ApuntadorA Nodo2[TipoEle]	-**Info()**. *Transformador.* Modifica la información del nodo por la que viene en el parámetro. -**PSig()**. *Transformador.* Modifica el apuntador al nodo siguiente por la dirección que viene en el parámetro. **-PAnt().** *Observador.*Regresa la dirección del nodo anterior. -**Info()**. *Observador.* Devuelve la información contenida en el nodo. -**PSig()**. *Observador.* Regresa la dirección del nodo siguiente.

Figura 1.95. Implementación de la clase Nodo2[TipoEle].

Lista2Enl[TipoEle] Clases: Entero, Lógico, ApuntadorA, TipoEle, Nodo2[TipoEle]		
1	**Superclases:** ninguna	-**n**: Número actual de elementos.
2	**Estructura**: privado: n : Entero+ = 0 pCab : ApuntadorA Nodo2[TipoEle] = Nulo pCursor : ApuntadorA Nodo2[TipoEle] = Nulo	-**pCab**: Apuntador al nodo especial. **-pCursor:** Apuntador al nodo actual. -**Lista2Enl()**. *Constructor.* Crea una lista vacía. **-~Lista2Enl().** *Destructor.* Destruye la lista. **-insLista()**. *Transformador.* Ingresa un nuevo elemento luego del nodo actual.
3	**Operaciones:** público: Lista2Enl() ~Lista2Enl() insLista(TipoEle: e) eliLista() conLista(): TipoEle vacíaLista(): Lógico numEle(): Entero+ cursorAlInicio() cursorAlProximo(): Lógico cursorAlAnterior(): Lógico despliegue()	**-eliLista()**. *Transformador.* Elimina el elemento después del nodo actual, si la lista no está vacía. -**conLista()**. *Observador.* Devuelve el elemento en el nodo actual, si existe. -**vacíaLista()**. *Observador.* Regresa Verdadero si la lista está vacía. -**numEle()**. *Observador.* Regresa el número actual de elementos en la lista. **-cursorAlInicio()**. *Transformador.* Coloca el cursor antes del primer nodo. **-cursorAlProximo().** *Transformador.* Avanza el cursor al próximo nodo, si fue posible regresa Verdadero. **-cursorAlAnterior().** *Transformador.* Retrocede el cursor al nodo anterior, si ello es posible regresa Cierto. **-despliegue().** *Observador.* Despliega el contenido de la lista.

Figura 1.96. Implementación de una lista según el método doblemente enlazado.

El destructor de la figura 1.98 debe devolver toda la memoria ocupada por los nodos que están asignados a la lista, incluyendo el nodo especial.

27/3/98 Lista2Enl() {pos: n = 0 ∧ pCab → PAnt() = Nulo ∧ pCab → PSig() = Nulo }		

1
2
	n, pCab = 0, nuevaDir Nodo2
	Si (pCab = Nulo) entonces
	  Despliegue "Memoria llena"
	sino
	  pCursor = pCab
	fsi
		-n, pCab, pCursor. Definidos en Lista2Enl.
		-nuevaDir. Regresa una dirección de memoria disponible del tipo Nodo2.

1	Lista2Enl[Entero] lis ⇒ lis.n = 0, lis.pCab ≠ Nulo y lis.pCursor ≠ Nulo	Crea la variable lis de enteros.

Figura 1.97. Constructor vacío. O(1).

27/3/98 ~Lista2Enl() {pre: n ≥ 0}		
1	(pCab ≠ Nulo) [pAux = pCab pCab = pCab → PSig() devuelvaDir pAux]	**-pCab.** Definido en Lista2Enl. **-PSig():** Definido en la clase Nodo2. **-pAux:** ApuntadorA Nodo2[TipoEle]. Variable auxiliar

Figura 1.98. Destruye la lista doblemente enlazada. O(n).

La función de inserción presentada en la figura 1.99, al igual que en la implementación con un solo enlace presenta las mismas instrucciones usadas para los tres casos de inserción en listas, estos son: inserción en vacío o al inicio, inserción en el medio e inserción al final.

27/3/98 insLista(TipoEle: e) {pre: e ≠ {TipoEleNoDef} ∧ n ≥ 0} {pos: n ≥ 0}		

1
2
	pAux = nuevaDir Nodo2[TipoEle]
	Si (pAux ≠ Nulo) entonces
	  pAux → PAnt(pCursor)
	  pAux → Info(e)
	  pAux → PSig(pCursor → PSig())
	  Si (pCursor → PSig() ≠ Nulo) entonces
	  pCursor → PSig() → PAnt(pAux)
	  fsi
	  pCursor → PSig(pAux)
	  n = n + 1
	sino
	  Despliegue "Memoria llena"
	fsi
		-pAux: ApuntadorA Nodo2[TipoEle]. Variable auxiliar.
		-Info(), PAnt(), PSig(): Definidos en la clase Nodo2.
		-n, pCursor: Definidos en Lista2Enl.
		-nuevaDir Nodo2[TipoEle]: Regresa una dirección de memoria disponible de tipo Nodo2[TipoEle].

1	lis = (0, →, →) (/, -8, /), e = 78 ⇒ lis = (1, →, →) (/, -8, →) (←,78, /)	Ingresa 78 en una lista vacía. El elemento no definido es –8.
2	lis = (2, →, →)(/,-8, →)(←,45, →)(←,34, /), e = 20 ⇒ lis=(3, →, →)(/,-8, →)(←,20, →)(←,45, →)(←,34, /)	Ingresa 20 al inicio de la lista, es decir después del nodo cabeza.

Figura 1.99. Inserta un nuevo elemento en la lista doblemente enlazada. O(1).

La figura 1.100 presenta el algoritmo de eliminación que también tiene una complejidad constante debido al uso del nodo especial. Se elimina el nodo después del nodo actual, si él existe.

Igual que en la inserción, el bloque de instrucciones necesario para efectuar la eliminación de un nodo es el mismo para los cuatro casos de eliminación, a saber: eliminación del primer nodo, de un nodo intermedio, del último nodo y del único nodo que queda en la lista.

27/3/98 eliLista ()		
{pre: n ≥ 0}		{pos: n ≥ 0}
1	Si (pCab → PSig() ≠ Nulo) entonces pAux = pCursor → PSig() Si (pCursor → PSig() → PSig() ≠ Nulo) entonces pCursor → PSig() → PSig() →PAnt(pCursor) fsi pCursor → PSig(pCursor → PSig( ) →PSig()) devuelvaDir pAux n = n - 1 sino Despliegue “Lista vacía” fsi	**-pAux:** ApuntadorA Nodo2[TipoEle]. Variable auxiliar. **-PAnt(), PSig():** Definidos en la clase Nodo2. **-n, pCab, pCursor**: Definidos en Lista2Enl. **-devuelvaDir:** Regresa la dirección a la memoria disponible.
1	lis=(0, →, →)(/, -8, /) ⇒ Lista vacía	No elimina
2	lis = (2, →, →)(/,-8, →) (←,45, →) (←,34, /) ⇒ lis = (1, →, →)(/,-8, →) (←,34, /)	Elimina 45 de la lista, siendo –8 el elemento no definido del TipoEle.

Figura 1.100. Elimina un elemento en la lista doblemente enlazada. O(1).

Las mismas consideraciones anteriores se hacen para el caso de una consulta en la lista enlazada simple de la figura 1.101. Dado el uso del nodo especial los tres casos de consulta: al nodo inicial, a un nodo intermedio y al nodo final, se realizan con la misma instrucción.

27/3/98 conLista(): TipoEle		
{pre: n ≥ 0}		{pos: n ≥ 0}
1	Si (pCab → PSig() = Nulo) entonces regrese pCab → Info() sino regrese pCursor → Info() fsi	**-Info(), PSig():** Definidos en la clase Nodo2. **-n, pCab, pCursor**: Definidos en Lista2Enl.
1	lis = (0, →, →)(/,-8, /) ⇒ -8	Devuelve el valor {TipoEleNoDef} = -8
2	lis = (2, →, →1er.nodo)(/,-8, →) (←,45, →) (←,34, /) ⇒ 45	Devuelve 45

Figura 1.101. Devuelve el contenido del nodo de la lista apuntado por el cursor. O(1).

27/3/98 vacíaLista(): Lógico		
{pre: n ≥ 0}		{pos: n ≥ 0}
1	Regresa (n = 0)	**-n.** Definido en Lista2Enl.
1	lis = (2, →, →)(/,-8, →)(←,34, →)(←,9, /) ⇒ Falso	No está vacía.
2	lis = (0, →, →)(/,-8, /) ⇒ Verdadero	La variable lis está vacía.

Figura 1.102. Verifica si la lista doblemente enlazada está vacía. O(1).

El resto de las funciones de la clase Lista2Enl[TipoEle] se pueden observar en las figuras desde la 1.102 hasta la 1.107, donde todas ellas tienen una complejidad constante a excepción de la función **despliegue()** cuya complejidad depende del tamaño de la lista.

27/3/98 numEle(): Entero+ {pre: n ≥ 0}		{pos: n ≥ 0}
1	Regresa n	**-n.** Definido en Lista2Enl.
1	lis = (2, →, →)(/,-8, →)(←,32, →)(←,88, /) ⇒ 2	Hay dos nodos en la lista.
2	lis = (0, →, →)(/,-8, /) ⇒ 0	La variable lis tiene 0 elementos.

Figura 1.103. Devuelve el número de elementos actuales en la lista. O(1).

27/3/98 cursorAlInicio() {pre: n ≥ 0}		{pos: n ≥ 0 ∧ pCursor = pCab }
1	pCursor = pCab	**-pCursor, pCab.** Definidos en la clase Lista2Enl.
1	lis = (2, →, →2do. nodo)(/,-8, →)(←,44, →)(←,88, /) ⇒ lis = (2, →, →)(/,-8, →)(←,44, →)(←,88, /)	Cursor al inicio de la lista.
2	lis = (0, →, →)(/,-8, /) ⇒ lis = (0, →, →)(/,-8, /)	Cursor al inicio de la lista.

Figura 1.104. Mueve el cursor al nodo inicial en la lista. O(1).

27/3/98 cursorAlProximo(): Lógico {pre: n ≥ 0}		{pos: n ≥ 0}

1
 	
2
	Si (pCursor → PSig() ≠ Nulo) entonces
	  pCursor = pCursor → PSig()
	  regrese Verdadero
	fsi
	regrese Falso
		-pCursor. Definido en Lista2Enl.
		-PSig(). Definido en Nodo2.

1	lis = (2, →, →1er. nodo)(/,-8, →)(←,56, →)(←,88, /) ⇒ lis = (2, →, →2do.nodo) (/,-8, →) (←,56, →) (←,88, /) | Verdadero	Coloca el cursor apuntando al segundo nodo de la lista.
2	lis = (0, →, →)(/,-8, /) ⇒ Falso	No hay próximo nodo.

Figura 1.105. Mueve el cursor al próximo nodo en la lista. O(1).

27/3/98 cursorAlAnterior(): Lógico {pre: n ≥ 0}		{pos: n ≥ 0}

1
 	
2
	Si (pCursor → PAnt() ≠ pCab ∧ pCursor ≠ pCab) entonces
	  pCursor = pCursor → PAnt()
	  regrese Verdadero
	fsi
	regrese Falso
		-pCab, pCursor. Definido en Lista2Enl.
		-PAnt(). Definido en Nodo2.

1	lis = (2, →, →2do. nodo)(/,-8, →)(←,56, →)(←,88, /) ⇒ lis=(2, →, →1er.nodo)(/,-8, →)(←,56, →)(←,88, /) | Cierto	Coloca el cursor apuntando al primer nodo de la lista.
2	lis=(2, →, →1er. nodo)(/,-8, →)(←,56, →)(←,88, /) ⇒ Falso	No hay nodo anterior.
3	lis = (0, →, →)(/,-8, /) ⇒ Falso	No hay nodo anterior.

Figura 1.106. Mueve el cursor al próximo nodo en la lista. O(1).

En conclusión, la complejidad de la clase Lista2Enl[TipoEle] es lineal O(n).

27/3/98 despliegue ()		
{pre: n ≥ 0}		{pos: n ≥ 0}

1	Si (n = 0) entonces
	  Despliegue “La lista está vacía”
	sino
	  Despliegue “Elementos de la lista”
	  pCursor = pCab → PSig()
	  [Despliegue “Elemento(“, i , “)=”,pCursor → Info()
	  pCursor = pCursor → PSig()] i = 1, n
	fsi
		-**pCursor, pCab.** Definidos en la clase Lista2Enl.
		-**Info(), PSig():** Definidos en la clase Nodo2.
		-**i**: Entero+. Subíndice del lazo.

1	lis = (0, →, →)(/,-8, /) ⇒ La lista está vacía	No hay elementos en la lista.
2	lis = (2, →, →)(/,-8, →) (←,45, →) (←,34, /) ⇒ Elementos de la lista | Elemento(1)=45 | Elemento(2)=34 |	Despliega el contenido de la lista con sus comentarios

Figura 1.107. Despliega los elementos actuales en la lista doblemente enlazada. O(n).

Lista circular doblemente enlazada:

Esta última implementación para listas es la más utilizada por ser la más segura debido a sus dos enlaces y al hecho de ser circular, lo cual permite el recorrido en ambos sentidos.

Como se necesitan dos enlaces se usa la misma clase Nodo2[TipoEle] ya definida. La implementación de esta clase llamada ListaCir2Enl[TipoEle] se presenta en la figura 1.108.

ListaCir2Enl[TipoEle] Clases: Entero, Lógico, ApuntadorA, TipoEle, Nodo2[TipoEle]		
1	**Superclases:** ninguna	-**n**: Número actual de elementos.
2	**Estructura**: privado: n : Entero+ = 0 pCab : ApuntadorA Nodo2[TipoEle] = Nulo pCursor : ApuntadorA Nodo2[TipoEle] = Nulo	-**pCab**: Apuntador al nodo especial. -**pCursor:** Apuntador al nodo actual. -**ListaCir2Enl ()**. *Constructor.* Crea una lista vacía. -**~ListaCir2Enl ().** *Destructor.* Destruye la lista. -**insLista()**. *Transformador.* Ingresa un nuevo elemento luego del nodo actual.
3	**Operaciones:** público: ListaCir2Enl () ~ListaCir2Enl () insLista(TipoEle: e) eliLista() conLista(): TipoEle vacíaLista(): Lógico numEle(): Entero+ cursorAlInicio() cursorAlFinal() cursorAlProximo() cursorAlAnterior() despliegue()	-**eliLista()**. *Transformador.* Elimina el elemento después del nodo actual, si la lista no está vacía. -**conLista()**. *Observador.* Devuelve el elemento en el nodo actual, si existe. -**vacíaLista()**. *Observador.* Regresa Verdadero si la lista está vacía. -**numEle()**. *Observador.* Regresa el número actual de elementos en la lista. -**cursorAlInicio()**. *Transformador.* Coloca el cursor antes del primer nodo. -**cursorAlFinal().** *Transformador.* Coloca el cursor en el último nodo (antes de la cabeza). -**cursorAlProximo().** *Transformador.* Avanza el cursor al próximo nodo. -**cursorAlAnterior().** *Transformador.* Retrocede el cursor al nodo anterior. -**despliegue().** *Observador.* Despliega el contenido de la lista.

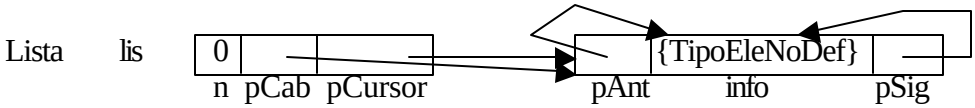
Figura 1.108. Implementación de una lista según el método doblemente enlazado circular.

Esta también utiliza el nodo especial que sirve para saber cuando se recorrió el círculo de nodos en cualquiera de los dos sentidos. Es la más segura porque si se pierde una de las direcciones almacenadas en un nodo, ésta se puede obtener, al recorrerse la lista en ambos sentidos hasta encontrar el corte y en el nodo siguiente o en el anterior se encuentra la dirección perdida, según la dirección perdida sea la del apuntador al anterior o al siguiente, respectivamente.

27/3/98 ListaCir2Enl () {pos: n = 0 ∧ pCab → PAnt() = pCab → PSig() = pCab }		
1 2	n, pCab = 0, nuevaDir Nodo2[TipoEle] Si (pCab = Nulo) entonces Despliegue "Memoria llena" sino pCursor = pCab pCab → PAnt(pCab) p Cab → PSig(pCab) fsi	**-n, pCab, pCursor.** Definidos en la clase Lista. **-nuevaDir.** Regresa una dirección de memoria disponible del tipo Nodo2[TipoEle]. **-PAnt(), PSig().** Definidos en la clase Nodo2.
1	ListaCir2Enl[Entero] lis ⇒ lis.n = 0, lis.pCab = lis.pCursor, pCab → pAnt = pCab → pSig = pCab	Crea la variable lis de enteros.

Figura 1.109. Constructor vacío. O(1).

Como se observa en la figura 1.109, el constructor vacío crea e inicia la lista, en este caso la condición de lista vacía es diferente a las anteriores, pues es una lista circular doblemente enlazada, por lo que hay que asignar los valores de incio a los apuntadores del nodo especial que es la cabeza de la lista. Esta condición se ilustra como sigue



La lista está vacía si los dos apuntadores al anterior y al siguiente del nodo especial apuntan o contienen la misma dirección del nodo especial.

El destructor de la figura 1.110 debe devolver toda la memoria ocupada por los nodos que están asignados a la lista, incluyendo el nodo especial. Es indiferente si para hacerlo se recorre la lista en un sentido o en el otro.

27/3/98 ~ListaCir2Enl () {pre: n ≥ 0}		
1 2	(pCab → PSig() ≠ pCab) [pAux = pCab → PSig() pCab → PSig(pCab→ PSig()→ PSig()) devuelvaDir pAux] devuelvaDir pCab	**-pCab.** Definido en ListaCir2Enl. **-PSig():** Definido en Nodo2. **-pAux:** ApuntadorA Nodo2[TipoEle]. Variable auxiliar

Figura 1.110. Destruye la lista circular doblemente enlazada. O(n).

En este destructor se eliminan todos los nodos de la lista y por último se elimina el nodo especial, ya que la lista es circular y no se pueden usar comparaciones con direcciones nulas porque no hay ninguna de esas en la lista.

27/3/98		
	insLista(TipoEle: e)	
{pre: e ≠ {TipoEleNoDef} ∧ n ≥ 0}		{pos: n ≥ 0}
1 2	pAux = nuevaDir Nodo2[TipoEle] Si (pAux ≠ Nulo) entonces pAux → PAnt(pCursor) pAux → Info(e) pAux → PSig(pCursor → PSig()) pCursor → PSig() → PAnt(pAux) pCursor → PSig(pAux) n = n + 1 sino Despliegue “Memoria llena” fsi	**-pAux:** ApuntadorA Nodo2[TipoEle]. Variable auxiliar. **-Info(), PAnt(), PSig():** Definidos en la clase Nodo2. **-n, pCursor**: Definidos en Lista2Enl. **-nuevaDir Nodo2[TipoEle]:** Regresa una dirección de memoria disponible de tipo Nodo2[TipoEle].
1	lis = (0, →, →) (↯, -8, ↶), e = 78 ⇒ lis = (1, →, →) (←1er.nodo, -8, →) (←,78, →cabeza)	Ingresa 78 en una lista vacía. El elemento no definido es –8.
2	lis = (2, →, →) (←1er.nodo,-8, →) (←,45, →) (←,34, →cabeza), e = 20 ⇒ lis = (3, →, →) (←1er.nodo, -8, →) (←,20, →) (←,45, →) (←,34, →cabeza)	Ingresa 20 al inicio de la lista, es decir después del nodo cabeza.

Figura 1.111. Inserta un nuevo elemento en la lista circular doblemente enlazada. O(1).

La figura 1.111 muestra la implementación de la función de inserción que tiene una complejidad constante por contener sólo instrucciones que no son lazos. Igual consideración se hace para la función que elimina el nodo actual y avanza el cursor al próximo nodo de la lista. Esta función se presenta en la figura 1.112.

27/3/98		
	eliLista ()	
{pre: n ≥ 0}		{pos: n ≥ 0}
1	Si (pCab → PSig() ≠ pCab ∧ pCab → PAnt() ≠ pCab) ent. pAux = pCursor → PSig( ) pCursor → PSig() → PSig( ) → PAnt(pCursor)) pCursor → PSig(pCursor → PSig( ) → PSig( )) devuelvaDir pAux n = n - 1 sino Despliegue “Lista vacía” fsi	**-pAux:** ApuntadorA Nodo2[TipoEle]. Variable auxiliar. **-PAnt(), PSig():** Definidos en la clase Nodo2. **-n, pCab, pCursor**: Definidos en Lista. **-devuelvaDir:** Regresa la dirección a la memoria disponible.
1	lis = (0, →, →) (↯, -8, ↶) ⇒ Lista vacía	No elimina
2	lis = (2, →, →) (←2do.nodo,-8, →) (←,45, →) (←,34, →cabeza) ⇒ lis = (1, →, →) (←1er.nodo, -8, →) (←,34, →cabeza)	Elimina 45 de la lista, siendo –8 el elemento no definido del TipoEle.

Figura 1.112. Elimina un elemento en la lista circular doblemente enlazada. O(1).

La consulta del elemento actual en la lista se puede observar en la figura 1.113. A continuación aparecen las funciones adicionales de la clase en las figuras que van desde la

1.114 a la 1.120. La complejidad total de la clase ListaCir2Enl[TipoEle] será lineal, pues tanto el **destructor** como la función **despliegue** tienen esa complejidad O(n).

27/3/98 conLista(): TipoEle {pre: n ≥ 0}		{pos: n ≥ 0}
1	regrese pCursor → Info()	**-Info():** Definidos en la clase Nodo2. **-pCursor**: Definidos en Lista.
1	lis = (0, →, →)(↳, -8, ↲) ⇒ -8	Devuelve el valor {TipoEleNoDef} = -8
2	lis = (2, →, →1er.nodo) (←2do.nodo,-8, →) (←,45, →) (←,34, →cabeza) ⇒ 45	Devuelve 45

Figura 1.113. Devuelve el contenido del nodo de la lista apuntado por el cursor. O(1).

27/3/98 vacíaLista(): Lógico {pre: n ≥ 0}		{pos: n ≥ 0}
1	Regresa (n = 0)	**-n.** Definido en Lista.
1	lis = (2, →, →) (←2do.nodo,-8, →) (←,34, →) (←,9, →cabeza) ⇒ Falso	No está vacía.
2	lis = (0, →, →) (↳, -8, ↲) ⇒ Verdadero	La variable lis está vacía.

Figura 1.114. Verifica si la lista doblemente enlazada está vacía. O(1).

27/3/98 numEle(): Entero+ {pre: n ≥ 0}		{pos: n ≥ 0}
1	Regresa n	**-n.** Definido en Lista.
1	lis = (2, →, →) (←2do.nodo,-8, →) (←,32, →) (←,88, →cabeza) ⇒ 2	Hay dos nodos en la lis ta.
2	lis = (0, →, →) (↳, -8, ↲) ⇒ 0	La variable lis tiene 0 elementos.

Figura 1.115. Devuelve el número de elementos actuales en la lista. O(1).

27/3/98 cursorAlInicio() {pre: n ≥ 0}		{pos: n ≥ 0 }
1	pCursor = pCab	**-pCursor, pCab.** Definidos en Lista.
1	lis = (2, →, →2do. nodo) (←2do.nodo,-8, →) (←,44, →) (←,88, →cabeza) ⇒ lis = (2, →, →1er.nodo) (←2do.nodo,-8, →) (←,44, →) (←,88, →cabeza)	Cursor al inicio de la lista.
2	lis=(0, →, →)(↳, -8, ↲) ⇒ lis = (0, →, →)(↳, -8, ↲)	Cursor al inicio de la lista.

Figura 1.116. Mueve el cursor al nodo inicial en la lista. O(1).

27/3/98 cursorAlFinal() {pre: n ≥ 0}		{pos: n ≥ 0 }
1	pCursor = pCab → PAnt()	**-pCursor, pCab.** Definidos en Lista. **-PAnt().** Definido en Nodo2.
1	lis = (2, →, →1er. nodo) (←2do.nodo,-8, →) (←,44, →) (←,88, →cabeza) ⇒ lis = (2, →, →2do.nodo) (←2do.nodo,-8, →) (←,44, →) (←,88, →cabeza)	Cursor al final de la lista.
2	lis=(0, →, →)(↳, -8, ↲) ⇒ lis = (0, →, →)(↳, -8, ↲)	Cursor al final de la lista.

Figura 1.117. Mueve el cursor al nodo final en la lista. O(1).

27/3/98 cursorAlProximo() {pre: n ≥ 0}		{pos: n ≥ 0}
1	pCursor = pCursor → PSig()	**-pCursor.** Definido en Lista. **-PSig().** Definido en Nodo2.
1	lis = (2, →, →1er. nodo) (←2do.nodo,-8, →) (←,56, →) (←,88, →cabeza) ⇒ lis = (2, →, →2do.nodo) (←2do.nodo,-8, →) (←,56, →) (←,88, →cabeza) | Verdadero	Coloca el cursor apuntando al segundo nodo de la lista. Hay próximo nodo.
2	lis = (0, →, →)(↯, -8, ↵) ⇒ Falso	No hay próximo nodo.

Figura 1.118. Mueve el cursor al próximo nodo en la lista. O(1).

27/3/98 cursorAlAnterior() {pre: n ≥ 0}		{pos: n ≥ 0}
1	pCursor = pCursor → PAnt()	**-pCursor.** Definido en Lista. **-PAnt().** Definido en Nodo2.
1	lis = (2, →, →2do. nodo) (←2do.nodo,-8, →) (←,56, →) (←,88, →cabeza) ⇒ lis=(2, →, →1er.nodo) (←2do.nodo,-8, →) (←,56, →) (←,88, →cabeza) | Verdadero	Coloca el cursor apuntando al primer nodo de la lista. Hay nodo anterior.
2	lis = (0, →, →)(↯, -8, ↵) ⇒ Falso	No hay nodo anterior.

Figura 1.119. Mueve el cursor al nodo anterior en la lista. O(1).

Hasta esta sección se han presentado las clases separadas según su tipo de implantación, en la próxima sección las clases serán relacionadas a través de la jerarquía es-un(a) o jerarquía de clases. Para esto se hará uso de la herencia simple, del encapsulamiento y del polimorfismo propio de la orientación por objetos.

1.18.5. Implantación de la jerarquía de clases para las estructuras lineales de datos

En esta sección se presentan las implementaciones de las clases ya vistas, pero siguiendo una estructura de clases diferente, basada en la jerarquía de clases mostrada en la figura 1.23. Para ello se toma la última clase presentada para las listas, la implementación de la figura 1.108 que se corresponde con una lista circular doblemente enlazada. Esta clase será extendida para hacerla más completa y poder soportar otras operaciones sobre las listas, como son: la unión, la intersección, la fisión, la comparación de igualdad de contenido, de diferencia de contenido, la copia y la asignación de listas. Su especificación se muestra en la figura 1.121.

Por ser la raíz de la jerarquía que se definió anteriormente, la clase Lista no tiene superclase, su estructura se mantiene igual, pero sus operaciones son modificadas y extendidas para soportar las operaciones ya mencionadas. La implementación propuesta puede ser vista en la figura 1.122. Se modificaron los nombres de las funciones sobre los elementos, por nombres genéricos para poder aplicárselas a cualquier otra de sus subclases sin pérdida de su sentido semántico.

Las funciones ya implementadas, como son: el constructor vacío (figura 1.109) se mantiene en esta clase, pero se le cambia en nombre por Lista(); igual para el destructor (figura 1.110) que se llama aquí ~Lista(); para la inserción (figura 1.111) se cambia el nombre a insertar(TipoEle: e); para la eliminación (figura 1.112) se usa el mismo algoritmo con el nombre de función eliminar(); la de consulta (figura 1.113) se renombra por consultar(); y por

último, el chequeo de si la lista está vacía (figura 1.114) se renombra por estaVacia(). El resto de las funciones se mantiene tal cual se implementaron en la sección anterior.

27/3/98 despliegue () {pre: n ≥ 0}		{pos: n ≥ 0}
1	Si (n = 0) entonces Despliegue "La lista está vacía" sino Despliegue "Elementos de la lista" pCursor = pCab → PSig() [Despliegue "Elemento(", i , ")=",pCursor → Info() pCursor = pCursor → PSig()] i = 1, n fsi	**-n, pCursor, pCab.** Definidos en la clase Lista. **-Info(), PSig():** Definidos en la clase Nodo2. **-i**: Entero+. Subíndice del lazo.
1	lis = (0, →, →) (↯, -8, ↯) ⇒ La lista está vacía	No hay elementos en la lista.
2	lis = (2, →, →) (←2do.nodo,-8, →) (←,45, →) (←,34, →cabeza) ⇒ Elementos de la lista | Elemento(1)=45 | Elemento(2)=34 |	Despliega el contenido de la lista con sus comentarios

Figura 1.120. Despliega los elementos actuales en la lista circular doblemente enlazada. O(n).

27/3/98 Especificación Lista[TipoEle]		
1	**Sintáctica:** creaLista()⟶ Lista, insertar(Lista,TipoEle)⟶ Lista, eliminar(Lista)⟶ Lista, consultar(Lista)⟶ TipoEle, estaVacía(Lista)⟶ Lógico, union(Lista, Lista)⟶ Lista, interseccion(Lista, Lista)⟶ Lista, fision(Lista, Lista)⟶ Lista, verificacion(Lista, Operador, Lista)⟶ Lógico, destruyeLista(Lista)⟶ .	**-creaLista()**: Crea una lista vacía. **-insertar()**: Ingresa un nuevo elemento a la lista según una posición especificada. **-eliminar()**: Elimina el elemento en la lista según una posición especificada. Si la lista está vacía no hace ninguna eliminación. **-destruyeLista()**: Destruye la lista. **-estaVacía()**: Regresa Verdadero si la lista está vacía. **-consultar()**: Devuelve el elemento en la lista seguún una posición especificada. Si la lista está vacía devuelve un valor especial que lo indica. **-union():** Une la primera lista con la segunda y la regresa en otra lista. **-interseccion():** Genera una nueva lista con los elementos comunes en ambas listas. **-fision():** Divide por la mitad la primera lista, regresando dos nuevas listas. **-verificacion():** Regresa Verdadero si la primera lista es {=, ≠} a la segunda lista.
2	**Declaraciones:** TipoEle: e, {TipoEleNoDef} Lista: ll	
3	**Semántica:** estaVacía(creaLista())=Verdadero estaVacía(insertar(creaLista(),e))=Falso consultar(creaLista())={TipoEleNoDef} eliminar(creaLista())=creaLista() union(creaLista(), creaLista())=creaLista() union(ll, creaLista())=union(creaLista(), ll)=ll interseccion(creaLista(), creaLista())=creaLista() interseccion(creaLista(), ll) = interseccion(ll, creaLista()) = creaLista() fision(creaLista(), creaLista()) = creaLista() verificacion(creaLista(), =, creaLista()) = Verdadero verificacion(creaLista(), ≠, creaLista()) = Falso	

Figura 1.121. Especificación completa del TAD Lista en TDSO.

Lista[TipoEle] Clases: Entero, Lógico, ApuntadorA, TipoEle, Nodo2[TipoEle]		
1	**Superclases:** ninguna	-**n**: Número actual de elementos.
2	**Estructura**: privado: n : Entero+ = 0 pCab : ApuntadorA Nodo2[TipoEle]= Nulo pCursor : ApuntadorA Nodo2[TipoEle] = Nulo	-**pCab**: Apuntador al nodo especial. -**pCursor:** Apuntador al nodo actual. -**Lista()**. *Constructores*. Crea una lista vacía o una lista copia de la lista enviada por parámetro. -**~Lista().** *Destructor.* Destruye la lista. -**insertar()**. *Transformador.* Ingresa un nuevo elemento luego del nodo actual.
3	**Operaciones:** público: Lista() Lista(Lista[TipoEle]) ~Lista() insertar(TipoEle: e) eliminar() consultar(): TipoEle estaVacía(): Lógico numEle(): Entero+ cursorAlInicio() cursorAlFinal() cursorAlProximo() cursorAlAnterior() despliegue() union(Lista[TipoEle]: l1): Lista[TipoEle] interseccion(Lista[TipoEle]: l1): Lista[TipoEle] fision(Lista[TipoEle]: l1): Lista[TipoEle] = (Lista[TipoEle]: l1): Lógico ≠ (Lista[TipoEle]: l1): Lógico = (Lista[TipoEle]: l1): Lista[TipoEle]	-**eliminar()**. *Transformador.* Elimina el elemento después del nodo actual, si la lista no está vacía. -**consultar()**. *Observador.* Devuelve el elemento en el nodo actual, si existe. -**estaVacía()**. *Observador.* Regresa Verdadero si la lista está vacía. -**numEle()**. *Observador.* Regresa el número actual de elementos en la lista. -**cursorAlInicio()**. *Transformador.* Coloca el cursor antes del primer nodo. -**cursorAlFinal().** *Transformador.* Coloca el cursor en el último nodo (antes de la cabeza). -**cursorAlProximo().** *Transformador.* Avanza el cursor al próximo nodo. -**cursorAlAnterior().** *Transformador.* Retrocede el cursor al nodo anterior. -**despliegue().** *Observador.* Despliega el contenido de la lista. -**union().** *Generador.* Genera una lista con la unión de las dos listas. -**interseccion().** *Generador.* Genera una lista con los elementos comunes de las dos listas. -**fision().** *Generador.* Genera dos listas con el contenido de la mitad de la lista original cada una. - **= ().** *Observador.* Regresa Verdadero si las dos listas son iguales en contenido. - **¹ ().** *Observador.* Regresa Verdadero si las dos listas son diferentes en contenido. - **= ().** *Transformador.* Asigna una copia de la lista a la lista resultante.

Figura 1.122. Implementación de la Lista[TipoEle] según el método doblemente enlazado.

Se pasa ahora a implementar las funciones adicionales. La primera que se presenta en la figura 1.123 es el constructor copia, que genera una lista nueva que contiene una copia de los elementos de la lista enviada por los parámetros. La lista enviada a través del parámetro no se modifica, por lo tanto el número de elementos se mantiene, el apuntador a nodo cabeza o encabezado se mantiene y también el cursor. La nueva lista se genera si hay memoria disponible, de lo contrario regresara una lista vacía o con los elementos que haya podido copiar.

27/3/98 Lista(Lista[TipoEle]: ll) {pos: n = ll.n ∧ la lista creada contiene lo mismo que ll }		
1 2	n, pCab, pCursor = 0, nuevaDir Nodo2[TipoEle], pCab Si (pCab = Nulo) entonces Despliegue "Memoria llena" sino pCab → PAnt(pCab) pCab → PSig(pCab) pRec = ll.pCab → PSig( ) (pRec ≠ ll.pCab) [pAux = nuevaDir Nodo2 Si (pAux = Nulo) entonces pRec = ll.pCab Despliegue "Memoria llena" sino pAux → Info(pRec → Info( )) pAux → PAnt (pCursor) pAux → PSig (pCab) pCursor → PSig (pAux) pCursor → PAnt (pAux) pCursor, n = pAux, n + 1 pRec = pRec → PSig ( ) fsi] fsi	**-n, pCab, pCursor.** Definidos en la clase Lista. **-nuevaDir.** Regresa una dirección de memoria disponible del tipo Nodo2[TipoEle]. **-PAnt(), PSig().** Definidos en la clase Nodo2. **-pRec, pAux:** ApuntadorA Nodo2[TipoEle]. Variables auxiliares para recorrer los nodos de la lista ll y para inserción de los nuevos elementos en la lista copia, respectivamente.
1	Lista[Entero] lis(ll), ll = (0, →, →) (↳, -8, ↲) ⇒ lis = (0, →, →) (↳, -8, ↲)	Crea la variable lis de enteros cuyo contenido es igual a ll.

Figura 1.123. Constructor copia para listas. O(n).

1.19. Implantación de las clases en C++

Cada una de las clases vistas en la sección 1.6 se traducen al lenguaje C++ y sus códigos se presentan en el apéndice A.

1.20. Ejercicios

Para cada uno de los enunciados iniciales de los problemas dados a continuación, realice su refinamiento según la técnica de desarrollo de sistemas de objetos (TDSO) y su análisis de eficiencia.

1. Realice una función especial para el TAD Pila[TipoEle] que elimine y devuelva el elemento igual al contenido de **y** según la implantación secuencial.
2. Suponga que las pilas **p1** y **p2** se almacenan juntas en un arreglo llamado **pilas** de tamaño **n**. La pila **p1** se almacena en la parte izquierda del arreglo y crece de izquierda a derecha, mientras que la pila **p2** se almacena en la parte derecha del arreglo y crece de derecha a izquierda. Asuma que $\mathbf{tope_k}$ señala el elemento tope de la pila $\mathbf{p_k}$ con **k** = {1, 2} y que **item** es la información que se desea meter o sacar de la pila indicada por **k**. Realice dos funciones, una para *sacar* el elemento **item** de la pila **k** y otra para *meter* el elemento **item** en la pila **k**.
3. Desarrolle una aplicación para determinar si una cadena de caracteres dada pertenece al grupo de las cadenas **W** que tienen la forma **w '&' w^r**, donde **w** es una cadena formada

por los símbolos '0' y '1' y $\mathbf{w}^r$ es la cadena inversa de **w**. Ejemplo: w = '0001011' entonces w^r = '1101000'.

4. Una vez realizado el ejercicio 1, escribir una aplicación para detectar si la cadena dada es de la forma **a '$' b '$' c '$' ... '$' z** donde **a,b,c, ... , z** pertenecen a las cadenas **W**.
5. Realice una función que vacíe una pila, pasando sus elementos a una cola respetando el orden de llegada de los elementos a la pila, esto es si el elemento llego de primero a la pila debe estar de primero en la cola, si llegó de segundo a la pila debe ser el siguiente en la cola, y así sucesivamente hasta dejar la pila vacía.
6. Realice una función asociada al TAD Cola[TipoEle] con una implantación enlazada simple circular para eliminar el i-ésimo elemento luego del inicio de la cola.
7. Desarrolle una función para el TAD Cola[TipoEle] con una implantación enlazada simple circular para concatenar dos colas.
8. Realice una implantación para el TAD Cola[TipoEle] según el método enlazado doble circular.
9. Construya dos funciones asociadas al TAD Cola[TipoEle] implantada según el método secuencial para:

✓ Insertar un nuevo elemento en la cola luego del p-ésimo elemento.
✓ Eliminar de la cola el elemento Z sin perder el orden en el que se encuentran dicho elementos.

10. Suponga que es un inversor y que adquiere un determinado número de acciones, por ejemplo 150, en cada uno de los meses de Enero, Abril, Septiembre y Noviembre, y que vende el mismo número de acciones en los meses de Junio y Diciembre. Los precios de cada acción para estos meses son:

Enero	Abril	Junio	Septiembre	Noviembre	Diciembre
Bs. 20	Bs. 30	Bs. 20	Bs. 40	Bs. 30	Bs. 20

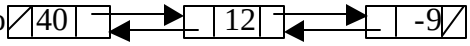
Determine el monto total de su ganancia o pérdida de capital utilizando:

✓ Contabilidad LIFO, es decir suponiendo que conserva sus certificados de valores en una pila
✓ Contabilidad FIFO, o sea suponiendo que conserva sus acciones en una cola

11. Un distribuidor de café recibe los pedidos de sus diferentes vendedores, los cuales son atendidos de acuerdo al orden de llegada y donde el número de pedidos por cada vendedor es variable. Los datos que se almacenan por pedido son: Número de pedido (numPed), nombre del cliente (nomCli), dirección del cliente (dirCli) y cantidad pedida en Kg. (canCli). Dado que el distribuidor posee una existencia limitada de café, éste deberá ir atendiendo los pedidos de cada vendedor hasta que se le agote la existencia. Se pide construir una aplicación que permita emitir un listado de los pedidos que serán satisfechos y además almacenar en un archivo de acceso secuencial, los pedidos que no fueron cubiertos por falta de existencia. Dicho archivo contendrá los datos siguientes por registro: Nombre del vendedor (nomVen), número de pedido (numPed), nombre del cliente (nomCli), dirección del cliente (dirCli) y cantidad pedida en Kg. (canCli).
12. Realice la especificación e implantación según el método enlazado simple de los TAD PilaDeColas[TipoEle], ColaDePilas[TipoEle] y ColaDeColas[TipoEle].
13. Realice la especificación e implantación según los métodos: secuencial, secuencial circular, enlazado simple, enlazado simple circular, enlazado doble y enlazado doble

circular de los TAD Dipolo[TipoEle], dipolo de entrada restringida (DipoloEntRest[TipoEle]) y dipolo de salida restringida (DipoloSalRest[TipoEle]).

14. Para el ejercicio anterior con el TAD Dipolo[TipoEle] según su implantación doblemente enlazada, pruebe sus rutinas en los casos dados a continuación:

    Asuma un estado actual de: Inicio  Fin

    Casos de prueba:

    1. insertar 4 por el inicio
    2. insertar 9 por el fin
    3. eliminar por el fin
    4. eliminar por el inicio
    5. eliminar por el fin
    6. eliminar por el inicio
    7. consultar por el inicio
    8. consultar por el fin
    9. eliminar por el inicio
    10. eliminar por el fin
    11. eliminar por el inicio
    12. insertar 5 por el fin

15. Un taller de reparación de automóviles posee un área de trabajo con capacidad para diez (10) carros colocados en fila y cuyo orden depende de la manera como llegan los clientes a solicitar el servicio. Cuando un carro ya fue reparado, se le avisa al cliente que pase a retirarlo. Cuando el cliente llega a retirar su carro que no está al incio de la fila, todos los carros que están delante de él deben ser sacados, éste lo retira del taller, y los automóviles deben ser colocados nuevamente en el mismo orden en que estaban, es decir **mantienen el orden** que había antes del último retiro. Cada vez que sale un carro, todos los otros carros detrás del retirado son movidos hacia delante, de tal manera que en todo momento los espacios vacíos estén al final de la fila del estacionamiento. Desarrolle la aplicación para que lea del archivo llamado **ordenes** las indicaciones siguientes:

    Llegada*placa del automóvil*nombre del cliente*teléfono del cliente* expresa la llegada de un nuevo carro, que será atendido dependiendo de su orden de llegada y de si hay o no cupo en la fila del estacionamiento.

    Salida*placa del carro* expresa que el carro está listo y que será retirado de la fila del estacionamiento por el cliente.

    Desplegar* indica que debe desplegarse en pantalla el estado actual de la fila del estacionamiento, asi como las estadísticas sobre los carros atendidos y ya reparados, los carros que no fueron atendidos por falta de cupo y los que están en reparación aún.

16. Un computador digital es una máquina que procesa programas. El procesador central toma los programas de una cola de programas, donde cada uno de ellos tiene asignada una prioridad entre 0 y 9, siendo 0 la prioridad más alta. Realice una aplicación para simular el computador insertando programas en la estructura de datos adecuada y eliminándolos de la misma basándose en su prioridad. Una vez obtenido el programa con más alta prioridad, debe rearreglarse la estructura para evitar que queden espacios vacíos dentro de la misma. Si dos o más programas tienen igual prioridad, pasará a procesarse aquel que esté de primero en la estructura. ¿Cuál es la estructura de datos adecuada para este sistema?.
17. Realice la especificación y la implantación de una función adicional para el TAD Lista[TipoEle] que intercambie el m-ésimo y el n-ésimo nodo de la lista.
18. Realice la especificación y la implantación de una función adicional para el TAD Lista[TipoEle] que invierta los nodos de una lista.
19. Realice la especificación y la implantación de una función adicional para el TAD Lista[TipoEle] que mezcle dos listas para producir una tercera lista. Considere la función para listas ordenadas y para las que no lo estén.

20. Dada una lista **ll** dividirla en dos listas según el nodo que contenga el valor contenido en **va**. Esto es, la primera lista contendrá los nodos de la lista original antes de **va** y la segunda contendrá el resto de los nodos incluso el nodo que contiene un valor igual al de **va**.
21. Dadas dos listas ordenadas ascendentemente por el campo cédula, obtenga la lista resultante de las dos anteriores, donde se encuentra toda la información de ambas listas sin repeticiones y adicionalmente incluya un campo extra calculado denominado sueldo neto (sueldoNeto), que se obtiene al restar las deducciones del sueldo base. Las lista contiene diferentes tipos de nodos, a saber:

    NodoLista1

cédula	nombre	dirección	teléfono	apuSig

    NodoLista2

cédula	sueldoBase	deducciones	apuSig

    En los casos en que no haya el compañero correspondiente en la otra lista, no debe ser tomado para pasar a la lista resultante.
22. Desarrolle una función para el TAD Lista[TipoEle] que intersecte dos listas desordenadas con respecto al campo **ca** y obtenga la lista intersección ordenada según dicho campo.
23. Una línea de autobuses que cubre varias rutas en el país desea tener una aplicación donde sus clientes puedan reservar sus puestos para un determinado viaje, así como cancelar sus reservaciones ya hechas. La información básica de cada viaje y para cada autobus designado en ese viaje es: la cédula del cliente (ced), su nombre (nom) y su puesto asignado (pa). Despliegue la lista de pasajeros de cada viaje-autobus. Escoja el método de implantación adecuado expresando las razones de su escogencia y las suposiciones que ud. tomó en cuenta para ello.
24. Especificar e implantar el TAD Polinomio3D para polinomios de tres (3) variables X, Y y Z. Utilizar la estructura de datos apropiada para almacenarlos con sus términos ordenados según el criterio siguiente, donde **p** y **q** son los dos polinomios:

    si $EX(p) < EX(q) \Rightarrow$ q va antes de p

    si $EX(p) = EX(q)$ y $EY(p) < EY(q) \Rightarrow$ q va antes de p

    si $EX(p) = EX(q)$ y $EY(p) = EY(q)$ y $EZ(p) < EZ(q) \Rightarrow$ q va antes de p

    si $EX(p) = EX(q)$ y $EY(p) = EY(q)$ y $EZ(p) = EZ(q) \Rightarrow$ COEF(p) + COEF(q)

    Considere dentro de las operaciones básicas del TAD las siguientes: suma (+), resta (-), multiplicación (*) , división (/), derivación con respecto de X (dx), con respecto de Y (dy), con respecto de Z (dz) y la evaluación en un punto 3D.
25. El siguiente problema se conoce como el *problema de José*. Dados un conjunto de nombres de soldados, el nombre del soldado donde se comenzará a contar (nomIni) y un número inicial de conteo (co), encontrar el soldado que irá a pedir refuerzos. Para ello, los soldados se colocan en círculo y se inicia el proceso buscando el nombre del soldado inicial en la estructura, a partir del cual se contará **co** soldados y el escogido será eliminado del círculo, escogiendo un nuevo **co** sobre la base del número que haya pensado el soldado eliminado, para continuar con el proceso de eliminación por conteo de los **co** soldados luego de él. El último soldado que quede en el círculo será el escogido para pedir refuerzos.
26. Un *sancocho de letras* es un pasatiempo que tiene un conjunto de letras donde se deben encontrar un conjunto de palabras dadas. La búsqueda de las palabras se realiza: (a) horizontalmente de izquierda a derecha, (b) horizontalmente de derecha a izquierda, (c)

verticalmente de arriba hacia abajo y (d) verticalmente de abajo hacia arriba. Nota: no se tomarán en cuenta los recorridos diagonales y todas las palabras dadas deben estar en el sancocho. Realice una aplicación que lea del archivo **sanDeLet** las líneas que forman el sancocho de letras seguidas de un ' * ' y luego las palabras a buscar separadas por blanco, forme una estructura de datos utilizando listas, encuentre las palabras y diga el sentido de cada una, (a, b, c ó d).

27. Una matriz cuyo contenido es en su mayoría ceros, se denomina *matriz esparcida o dispersa*.

$$\begin{pmatrix} 0 & 3 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 8 \\ 0 & 0 & 0 & 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 3 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 & 0 \\ 0 & 0 & 9 & 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 & 7 & 0 \\ 0 & 1 & 0 & 0 & 5 & 0 & 0 & 0 & 7 & 0 & 0 & 0 & 9 & 0 & 0 \end{pmatrix}$$

Una matriz de 6x15 = 90 localidades de memoria para 17 valores diferentes de cero. La mejor forma de representar este tipo de matrices sin perder mucho espacio de almacenamiento es a través de listas de múltiples enlaces, cuyos nodos tiene la forma:

fila	columna	valor	apuFil	apuCol

donde: **fila** es la fila donde está el elemento, **columna** es la columna del elemento, **valor** es el valor del elemento, **apuFil** es el apuntador al próximo nodo que contiene un elemento diferente de cero en esa fila, si existe; y **apuCol** es el apuntador al próximo nodo que contiene un elemento diferente de cero en esa columna, si existe. Para permitir la entrada por fila y por columna se mantienen dos vectores **vecFil** y **vecCol**. Realice una aplicación que transforme la matriz esparcida a su representación de listas y permita la búsqueda de algún elemento en ella bajo esta última representación.

1.21. Referencias bibliográficas

N. Dale y S. Lilly. *Pascal y estructuras de datos.* McGraw-Hill. 1989.

D. Knuth. The art of computer programming. Fundamental algorithms. Vol. 1. Addison-Wesley. 1973..

J. Tremblay y R. Bunt. Introducción a la ciencia de las computadoras. Enfoque algorítmico. McGraw-Hill. 1982.

J. Tremblay y P. Sorenson. *An introduction to data structures with applications.* McGraw-Hill. 1984.

A. Tenenbaum y M. Augestein. *Estructuras de datos en Pascal.* Prentice-Hall. 1983.

M. Weiss. *Estructuras de datos y algoritmos.* Addison-Wesley. 1995.

N. Wirth. *Algoritmos y estructuras de datos.* Prentice-Hall. 1987.