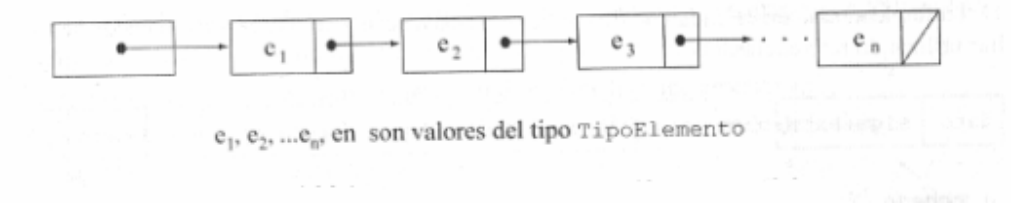
Lista enlazada o encadenada

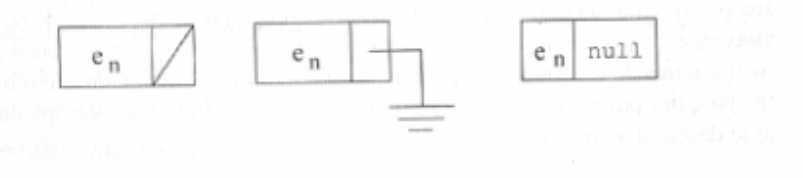
Es una estructura de datos dinámica, que crece y se contrae a medida que se ejecuta el programa.

Una Lista enlazada tiene las siguientes características:

- Estructura de datos ***dinámica***
- Conocida como ***Linked List***
- Es una ***colección de elementos*** (denominados nodos), cada uno de ellos ***conectado al siguiente por un enlace*** o ***referencia***.
- Los ***nodos*** se componen de ***dos partes (campos):*** La primera parte ***contiene la información*** y es un valor de tipo genérico (llamado Dato, TipoElemento, Info, etc.) y ***la segunda parte es una referencia*** (llamado enlace o siguiente) que apunta (enlaza) ***al siguiente elemento de la lista.***
- Los ***enlaces*** se representan ***por flechas*** para facilitar la comprensión de la conexión entre los nodos e indicar que el enlace tiene la dirección en memoria del siguiente nodo.
- Los ***enlaces*** sitúan los ***nodos en una secuencia***.



Distintas representaciones del último nodo



Clasificación de listas enlazadas

Hay distintas categorías:

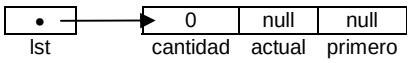
- ***Listas simplemente enlazadas***: Cada ***nodo contiene un único enlace*** que lo conecta ***al nodo siguiente***. Además hay una referencia al primer nodo y una al actual. Puede haber eventualmente una referencia al último. La lista es ***eficiente en recorridos directos*** (adelante).
- ***Listas simplemente enlazadas con nodo cabecera y/o centinela.*** Es igual a la anterior pero con dos nodos ficticios, uno al principio y otro al final. A costa de un pequeño uso de memoria extra se mejora la performance de algunas de las operaciones como veremos más adelante.

- ***Listas doblemente enlazadas***: Cada ***nodo contiene dos enlaces, uno a su nodo predecesor y otro a su nodo sucesor***. La lista es ***eficiente*** tanto en ***recorrido directo (adelante) como inverso (atrás).***
- ***Lista circular simplemente enlazada***: ***el último elemento (cola) se enlaza al primer elemento (cabeza)*** de tal modo que la lista ***puede ser recorrida*** en forma ***circular (como un anillo).***
- ***Lista circular doblemente enlazada***: ***el último elemento (cola)*** se ***enlaza al primer elemento (cabeza) y viceversa***. Se puede ***recorrer en forma circular (anillo) tanto de dirección directa (adelante) como inversa (atrás).***

Lista simplemente encadenada

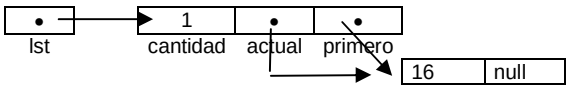
El primer nodo es el apuntado ***primero.*** El último es el nodo que tiene como referencia ***null***. Una lista se define con dos variables de tipo Nodo que llamaremos ***primero*** que apunta al primer nodo y la otra ***actual*** que apunta al nodo actual y una variable entera que indica la cantidad de elementos de la lista. (algunas implementaciones también utilizan una variable Nodo que apunta al último elemento de la lista). La lista ***se recorre del primer nodo al ultimo nodo***. ***Una lista vacia no contiene nodos, y se representa con el puntero primero con null y cantidad con valor cero .***

La lista vacia (lst = <>) se representa



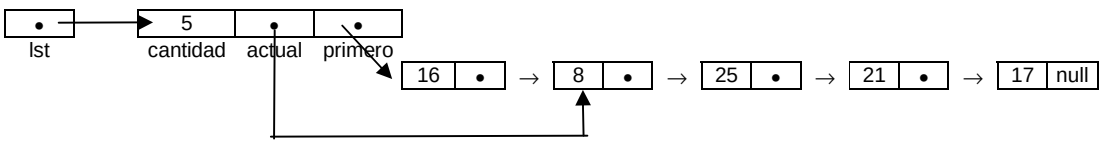
Una lista con un único elemento siendo éste el actual.

lst = <**16**>



Una lista con 5 elementos enteros siendo el segundo el actual.

lst = <16, **8**, 25, 21, 17>



ALICIA GIOIA
CLAUDIA LITVAK

Con esta representación del TAD Lista, la codificación de los métodos **irSiguiente( )**, **irPrimero()**, **verActual ( )**, **longitud(), finLista()** y **esVacia( )** son triviales siendo todas O(1).

Recordemos, en primer lugar, la implementación de la clase **Nodo.**

```
public class Nodo {
    Object dato;
    Nodo sig;

   public Nodo(){
      }
    public Nodo(Object x){
          dato = x;
      }
    public Nodo(Object x, Nodo ref){
          dato = x;
          sig = ref;
      }
}
```

```
public class ListaD implements Lista
{
   private Nodo primero;
   private Nodo actual;
   private int cantidad;

// ListaD construye una lista vacia
   public ListaD()      {
     primero = null;
     actual = null;
     cantidad = 0;
   }

// Métodos O(1)
  public void irSiguiente(){
        actual = actual.sig;
    }

    public void irPrimero(){
        actual = primero;
    }

    public Object verActual(){
        return actual.dato;
    }

    public int longitud(){
       return cantidad;
   }

   public boolean esVacia(){
       return cantidad == 0;
   }
```

```
  public boolean finLista(){
     return actual == null;
  }
// otros métodos a analizar a continuación

}
```

El resto de las operaciones son algo más complejas y vamos a analizarlas a continuación:

Colocar el actual en el último elemento de la lista

Debemos recorrer la lista desde el **actual** hasta el último elemento, el cual lo identificamos porque su campo **sig** es **null**. Es un método del orden O(n).

```
public void irUltimo(){
     while (actual.sig != nill)
         actual = actual.sig;
}
```

Colocar el actual en el elemento anterior del mismo
Este método es algo más complejo pues no sabemos la dirección del anterior del **actual**. Debemos recorrer la lista desde el principio hasta el anterior del actual. El método es O(n).

```
    public void irAnterior(){
         Nodo aux, ant;
         ant = null;
         aux = primero;
         while(aux != actual){
             ant = aux;
             aux = aux.sig;
         }
         actual = ant;
    }
```

Colocar el actual en una determinada posición de la lista
Por ejemplo supongamos que el actual esté en la posición 3 y lo queremos en la posición 5.
Antes
lst = <e_1, e_2, **e_3**, e_4, e_5, e_6, e_7, e_8>

Después
lst = <e_1, e_2, e_3, e_4, **e_5**, e_6, e_7, e_8>

```
    public void irA(int pos){
        actual = primero;
        int posActual = 1;
        while(posActual < pos){
            posActual++;
```

```
            actual = actual.sig;
        }
    }
```

Retornar la posición del actual

Debemos retornar el número de orden del actual.

```
public int verPosActual(){
      Nodo aux = primero;
      int pos = 1;
      while(aux != actual){
          aux = aux.sig;
          pos++;
      }
      return pos;
   }
```

Inserción de un elemento en una lista enlazada

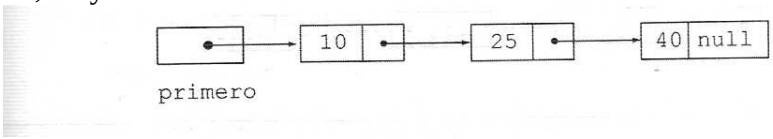
El nuevo elemento que se desea incorporar a una lista se puede insertar de distintas formas, según la posición o punto de inserción:

- En la cabeza de la lista (elemento primero)
- Después del elemento actual.
- Antes del elemento actual.

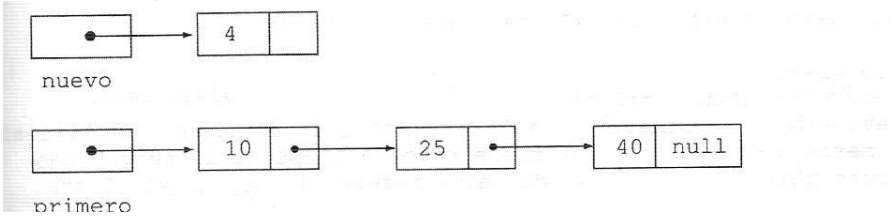
Insertar un nuevo elemento al principio de la lista

- Es el lugar más fácil y más eficiente donde insertar.
- Crear un nodo e inicializar el campo dato al nuevo elemento. La referencia del nodo creado se asigna a nuevo, una variable local del método.
- Hacer que el campo enlace del nuevo nodo apunte al primero de la lista original.
- Hacer que primero apunte al nodo que se ha creado

Ejemplo: Insertar un elemento 4 al principio de una lista enlazada que contiene 3 elemento 10, 25 y 40.



Paso 1



Código Java

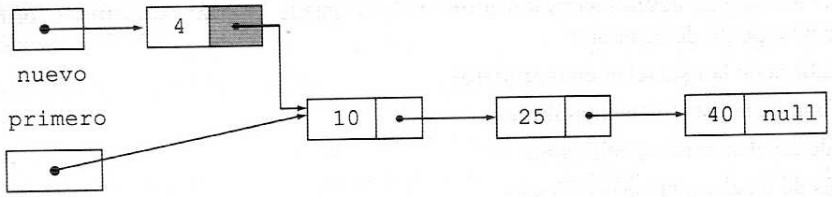
```
Nodo nuevo;
nuevo = new Nodo(entrada); // asigna un nuevo nodo
```

Paso 2

El campo `enlace` del nuevo nodo apunta al nodo `primero` actual de la lista.

Código Java

```
nuevo.enlace = primero
```

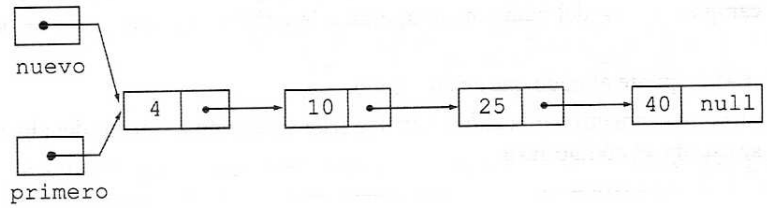


Paso 3

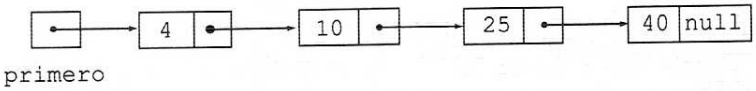
Se cambia la referencia de `primero` para que apunte al nodo creado; es decir, `primero` apunta al mismo nodo al que apunta `nuevo`.

Código Java

```
primero = nuevo;
```



En este momento, el método de insertar termina su ejecución, la variable local `nuevo` desaparece y sólo permanece la referencia al primer nodo de la lista: `primero`.



Caso particular

El método para insertar un elemento al principio de la lista también actúa correctamente si se trata de añadir un primer nodo o elemento a una lista vacía. En ese caso, como ya se ha comentado, primero apunta a null, y termina apuntando al nuevo nodo de la lista enlazada.

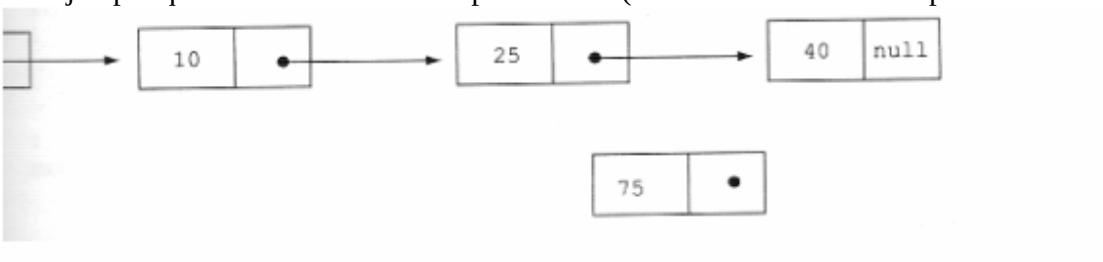
Insertar después del elemento actual

Tenemos que considerar dos casos:

1. Si la lista está vacía
2. La lista ya tiene elementos

El primer caso ya lo le hemos tratado en el punto anterior. Veamos el segundo caso.

Por ejemplo queremos insertar 75 después del 25 (la variable **actual** está apuntando a 25)

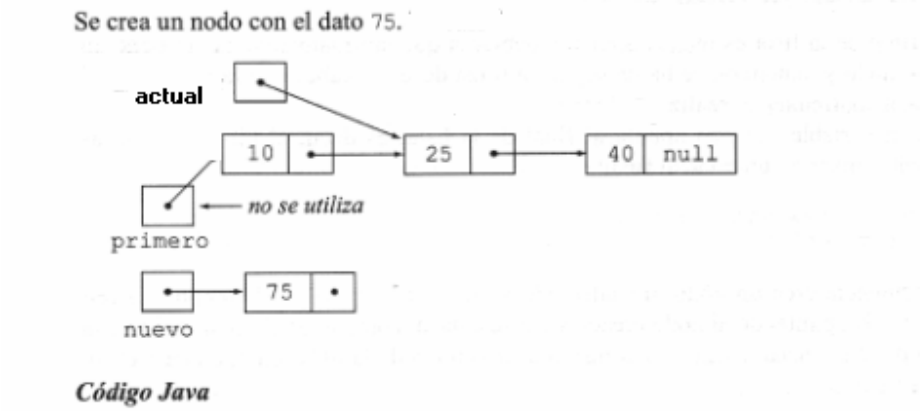


El algoritmo para la operación ***insertarDespues*** del actual requiere de las siguientes etapas:

1. ***Crear un nodo con el nuevo elemento*** y el campo ***sig*** en ***null***. La referencia al nodo se agina a nuevo.
2. Hacer que el campo ***sig*** del nuevo ***nodo*** apunte al nodo ***siguiente del actual***, ya que el nodo creado se ubicará justo ***antes de éste.***
3. La variable referencia del actual tiene la dirección del ***nodo siguiente*** (el 40 del grafico) y eso exige hacer que ***actual.sig*** apunte al nodo creado.
4. Finalmente actual debe apuntar al nuevo nodo y cantidad incrementarse en 1.

Las etapas y el código quedan como sigue:

Etapa 1

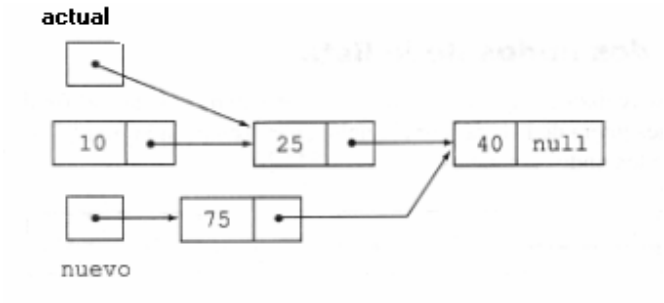


Código Java

```
Nodo nuevo = new Nodo(x);
```

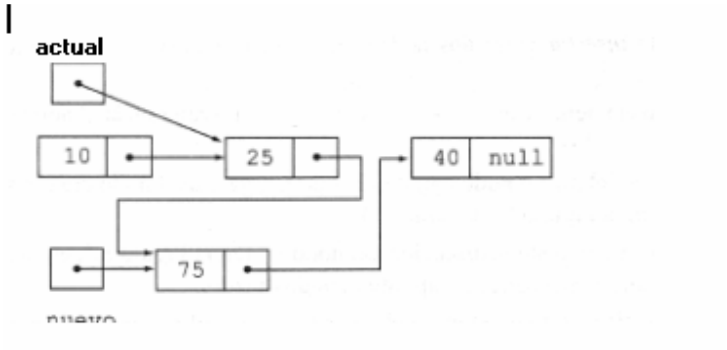
Etapa 2

El campo **sig** del nodo creado debe apuntar al siguiente del actual.



Código Java
```
nuevo.sig = actual.sig;
```

Etapa 3
El campo **sig** de **actual** debe apuntar al nuevo nodo



Código Java
```
actual.sig = nuevo;
```

Etapa 4
Finalmente **actual** debe apuntar al nuevo nodo y cantidad incrementarse en 1.

Codificación de insertar después del actual

```
public void insertarDespues(Object x){
    Nodo nuevo = new Nodo(x);
    if (esVacia()){
        primero = nuevo;
    }
    else{
        nuevo.sig = actual.sig;
        actual.sig = nuevo;
    }
    actual = nuevo;
    cantidad++;
 }
```

Insertar antes del elemento actual
Insertar el nuevo elemento antes del actual es un poco más complicado. Esto se debe que al conocer la dirección del **actual** es fácil conocer las direcciones de los siguientes pero no la de los nodos anteriores. Debemos considerar dos casos:

1. Lista vacía o el actual es el primero
2. El actual no es el primero.

El primer caso ya lo hemos analizado. El segundo caso tiene varias soluciones.
Primera solución

- ir al anterior del actual
- insertar después de éste

Esta solución es O(n).

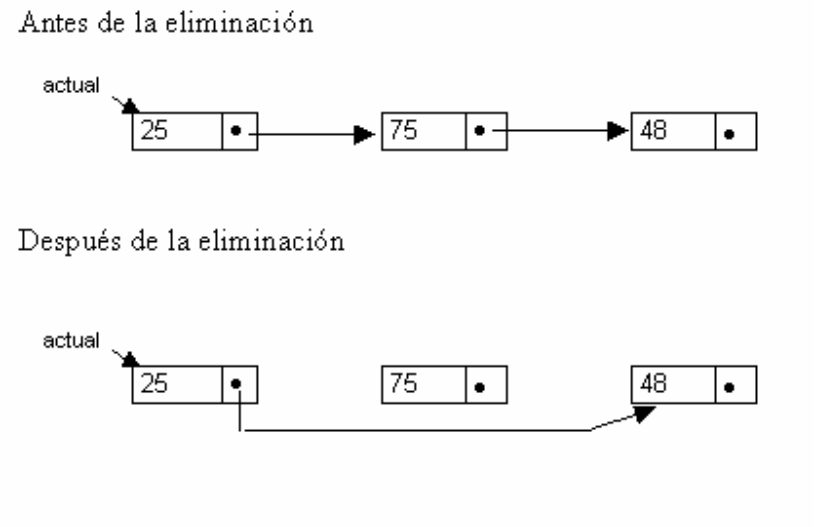
Segunda solución
Usar un nodo doble es decir un nodo con un apuntador al siguiente y otro con un apuntador al anterior (esto es en realidad una lista doblemente encadenada). Esta implementación complica las inserciones y eliminaciones puesto que hay que actualizar dos referencias y además usa más memoria.

Tercera solución

- Insertamos un nuevo nodo detrás del actual.
- Copiamos el campo dato del actual en el dato del nuevo.
- Copiamos la nueva información en el campo dato del actual.
- Aumentamos cantidad en 1.

Inserción al final de la lista
La ***inserción al final de la lista es menos eficiente***, debido a que, normalmente, no se tiene un puntero al último nodo, y entonces, se ha de recorrer la lista hasta el último nodo para, a continuación, realizar la inserción.
Una vez que la variable ***actual*** apunta al final de la lista, es decir, al último nodo, se realiza ***insertarDespues()***. Este es un método O(n).
Esto se puede simplificar si además de primero y actual tenemos una variable que nos referencie al último nodo.

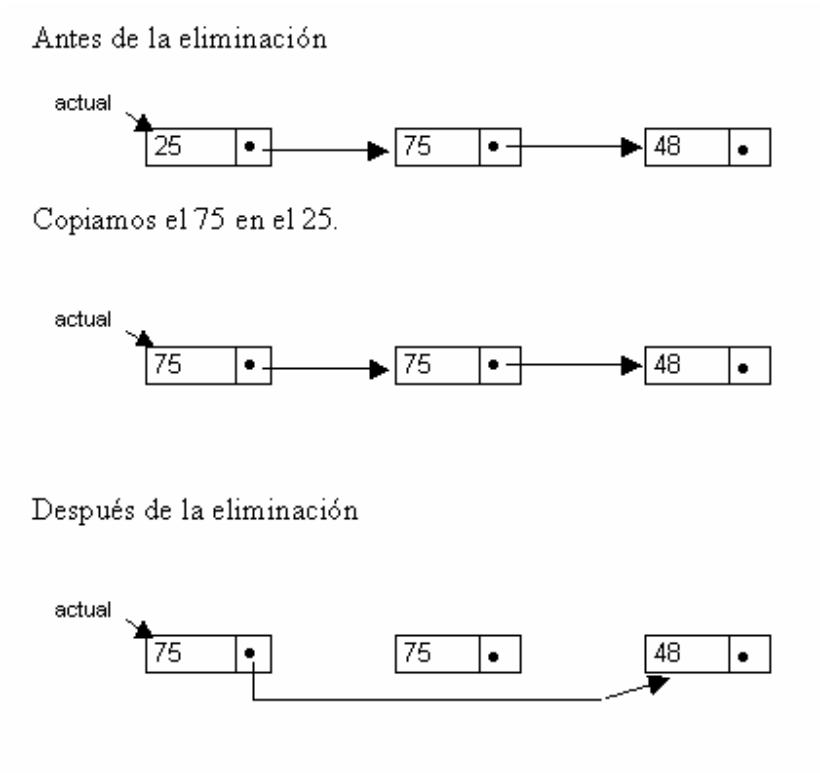
Eliminación del nodo siguiente de actual
La eliminación del **nodo siguiente del actual** es muy simple. Supongamos que **actual** apunte al 25 y queremos eliminar 75.



Código Java

```
actual = actual.sig;
```

El problema es que queremos eliminar el **actual**. Podríamos colocar el **actual** en anterior y luego eliminamos el siguiente del anterior que es el **actual**. Otra vez tenemos un método el orden de n.
Otra forma es hacer algo similar al método **insertarAntes().** Copiamos el contenido del siguiente en el **actual** y eliminamos el siguiente.



Código Java

```
actual.dato = actual.sig.dato;
actual.sig = actual.sig.sig;
```

El problema que este algoritmo no funciona si **actual** es el último pues **actual.sig** es **null** y por lo tanto **actual.sig.dato** va a dar un error en tiempo de ejecución. Por consiguiente el algoritmo de eliminación debe considerar como caso particular si el elemento a eliminar es el último.

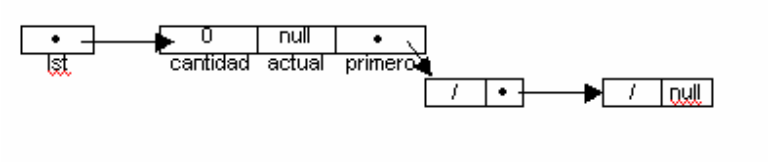
```
    public void eliminar(){
         if(actual.sig != null){
             actual.dato = actual.sig.dato;
             actual.sig = actual.sig.sig;
         }
         else {
             irAnterior();
             actual.sig =null;
         }
         cantidad--;
     }
```

Lista simplemente encadenada con nodo cabecera y nodo centinela

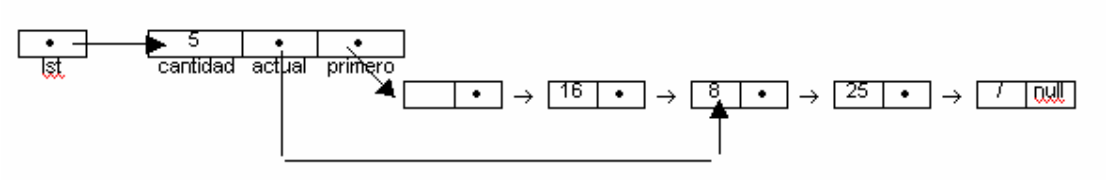
Para evitar tantos casos particulares en las inserciones y/o eliminaciones en una lista podemos, a costa de un pequeño gasto de memoria, agregar dos nodos ficticios (sin información importante) uno denominado **cabecera** que es apuntado por primero y otro al final denominado **centinela**.

Representación gráfica de una lista simplemente encadenada con nodo cabecera y nodo centinela

La lista vacía (lst = <>) se representa



La lista lst = <16, **8**, 25>



La clase **ListaDCC** implementa el TAD Lista usando encadenamiento simple con nodo cabecera y con nodo centinela. Los atributos y constructora son:

```
public class ListaDCC implements Lista
```

```
{
   private Nodo primero;
   private Nodo actual;
   private int cantidad;

// ListaD construye una lista vacia
   public ListaDCC()    {
     primero = new Nodo();   // nodo cabecera
     primero.sig = new Nodo();   //nodo centinela
     actual = null;
     cantidad = 0;
   }
```

Ejercicio
Codificar los métodos del TAD LISTA con esta representación.

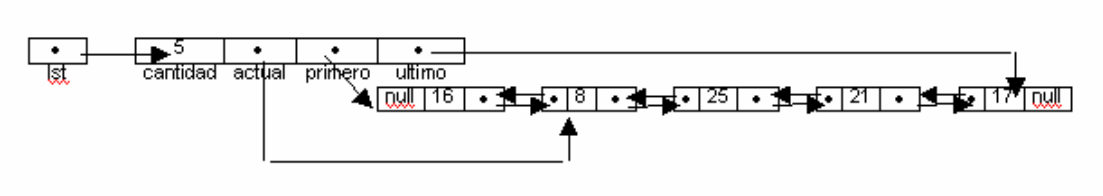
Lista doblemente encadenada
Cada ***nodo contiene dos enlaces, uno a su nodo predecesor y otro a su nodo sucesor***. La lista es ***eficiente*** tanto en ***recorrido directo (adelante) como inverso (atrás).***

En este caso debemos definir un nodo doble

```
public class NodoDoble {
    Object dato;
    Nodo sig, ant;

   public Nodo(){
      }
    public Nodo(Object x){
          dato = x;
      }
    public Nodo(Object x, Nodo refSig, Nodo refAnt)  {
          dato = x;
          sig = refSig;
          ant = refAnt;
      }
}
```

La representación gráfica es:



La clase **ListaDE** implementa el TAD Lista usando encadenamiento doble. Los atributos y constructora son:

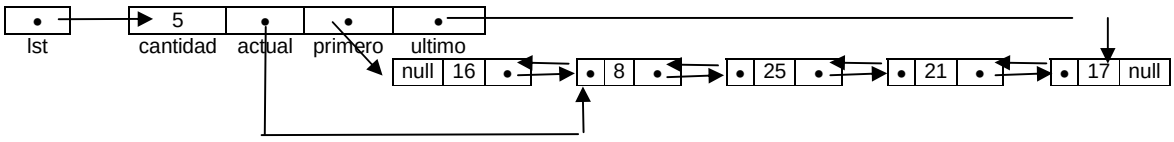
```
public class ListaDE implements Lista
{
   private Nodo primero;
   private Nodo actual;
   private Nodo ultimo;
   private int cantidad;

// ListaD construye una lista vacia
   public ListaDE()     {
     primero = null;
     ultimo = null;
     actual = null;
     cantidad = 0;
   }
```

Claramente los métodos irAnterior() y irUltimo() ahora don O(1) a costa de mayor espacio y un grado mayor de complejidad en las inserciones y eliminaciones ya que se deben actualizar mayor cantidad de referencias.

Ejercicio
Codificar insertarAntes(), insertarDespues() y elminar() utilizando esta representación.

La implementación doblemente encadenada también podemos agregarle nodo cabecera y centinela.



ALICIA GIOIA
CLAUDIA LITVAK