Estructuras lineales: Listas

Definiciones y conceptos básicos

Se define una lista ***como una secuencia de cero o más elementos de un mismo tipo***.
El formalismo escogido es

$< e_1, e_2, \ldots, e_n>$

La ***posición*** de un elemento dentro de una lista es el lugar ocupado por dicho elemento:

$$\begin{array}{l} \quad 1 \quad 2 \quad\ \ i \quad\ \ n \\ < e_1, e_2, \ldots, e_i \ldots, e_n> \end{array}$$

El ***sucesor*** de un elemento dentro de la lista es aquel que ocupa la siguiente posición. El único elemento de una lista que no tiene sucesor es el último. El ***antecesor*** es el elemento que ocupa la posición anterior. Todos los tienen salvo el primer elemento.

Un proceso muy importante en las estructuras de datos es su ***recorrido***: Este consiste en ***pasar exactamente una vez sobre cada uno de los elementos***, ya sea para encontrar un elemento en particular o para establecer alguna característica de la estructura. Una lista generalmente se recorre desde el primer elemento al último.

Ejercicio 1:

Para la lista representada mediante el objeto abstracto <1,3,5,7,11,13,17,19>:

- .... es el primer elemento, ... es el ultimo y <3,5,7,11,13,17> es el resto.
- La longitud de la lista es .....
- El sucesor de 11 es … y el sucesor de 5 es ….
- El antecesor de 19 es ….

Definiciones sobre listas:

- Dos listas list1 y list2 son ***iguales*** si ambas estructuras tienen el mismo numero de componentes y sus elementos son iguales uno a uno. En particular dos listas vacías son iguales.
- Dos listas list1 y list2 son ***semejantes*** si tienen los mismos elementos aunque estén en diferente orden. Si existe un elemento repetido en list1 debe aparecer el mismo numero de veces en list2.
- Una lista list2 es una ***sublista*** de una lista list1 si todos los elementos de lis2 se encuentran en list1, consecutivos y en el mismo orden (También se puede decir que lis2 ocurre en list1). En particular una lista vacía es una sublista de cualquier otra lista y una lista es sublista de si misma.
- Una lista list2 esta ***contenida*** en una lista list1, si todos los elementos de list2 están en list1, aunque sea en diferente orden.
- Una lista list1 es ***ordenada*** si los elementos contenidos respetan una relación de orden ≤ definida sobre ellos de acuerdo a la posición que ocupan:

  $e_i \leq e_j$ para todo $i \leq j$

  o sea list1 = $< e_1, e_2, \ldots, e_n>$ es ordenada si para todo i ($1 \leq i \leq n-1$) se cumple $e_i \leq e_{i+1}$.

Ejercicio 2:

Para las listas representadas mediante los objetos abstractos
list1 = <1,3,5,7, 9>
list2= < 1,2,3,4,5,6,7,8,9>
list3= <5,6,7>
list4= <5,6,7>
list5= <6,5,7>

Definir las relaciones que se cumplen:

ALICIA GIOIA
CLAUDIA LITVAK

- list3 es ….. a list4
- list3 es ……. a lis4 y list5
- …..., ……., ……., …… son listas ordenadas con la relación de orden ≤ definida sobre los números naturales .
- …… es una sublista de list2.
- …… esa contenida en list2

Especificación del TAD Lista

Existen muchos diseños abstractos posibles para manejar los objetos abstractos anteriormente descriptos. Uno de ellos es el siguiente:
Se define el elemento ***actual*** o ***ventana*** de una lista como el lugar de la secuencia sobre la cual se van a realizar operaciones que se apliquen al objeto abstracto.

1 2 **i** n
< e_1, e_2, …, $\mathbf{e_i}$ …., e_n>

Si el valor de actual es menor que 1 o mayor que cantidad, decimos que el actual está indefinido.

Los métodos son:

Constructora:

```
Lista()
```

Modificadoras:

```
insertarAntes( )
insertarDespues( )
eliminar( )
irSiguiente( )
irAnterior( )
irA(pos)
irPrimero()
irUltimo()
```

Analizadoras

```
verActual ( )
verPosActual( )
longitud( )
esVacia( )
finLista()
```

Especificación del TAD Lista

Nombre del TAD: Lista

Constructoras:

- **Lista:** - → **Lista**

ALICIA GIOIA
CLAUDIA LITVAK

/* Crea una lista vacía*/

pre: -

post: Lista lst = < >

- **insertarDespues: Lista X Elemento → Lista**

/* Agrega un elemento después de la ventana y el nuevo elemento pasa a ser el actual*/

pre: (Lista lst = <> ∨ lst = <x_1, ..., x_i,…., x_n>) y Elemento elem

post: lst = <elem> ∨ lst = <x_1, ..., x_i, elem …., x_n>)

- **insertarAntes: Lista X Elemento → Lista**

/* Agrega un elemento antes de la ventana y el nuevo elemento pasa a ser el actual*/

pre: (Lista lst = <> ∨ lst = <x_1, ..., x_i,…., x_n>) y Elemento elem

post: lst = <elem> ∨ lst = <x_1, ..., elem, x_i,…., x_n>)

- **eliminar: Lista →Lista**

/* Elimina el elemento que se encuentra en la ventana y el elemento siguiente pasa a ser el actual; si el elemnto eliminado es el último, el actual queda indefinido*/

pre: Lista lst != <>; (lst = <x_1, ... x_{i-1}, x_i, x_{i+1} , …., x_n>) ∨ (lst = <x_1, ..., x_i,…. x_{n-1}, x_n>)

post: lst = (lst = <x_1, ... x_{i-1}, x_{i+1} , …., x_n>) ∨ (lst = <x_1, ..., x_i,…., x_{n-1}>)

- **irSiguiente : Lista →Lista**

/* Avanza la ventana una posición; si la ventana está en el último queda indefinida*/

pre: Lista lst = <x_1, ..., x_i,…., x_n> v lst = <x_1, ..., x_i,…., x_n> y lst != <>

post: lst = <x_1, ..., x_{i+1},…., x_n> v lst = <x_1, ..., x_i,…., x_n>

- **irAnterior : Lista →Lista**

/* Retrocede la ventana una posición*/

(pre: lst = LST = <x_1, ..., x_i,…., x_n>)

(post: (LST = <x_1, ..., x_i,…., x_n>, lst = <x_1, ..., x_{i-1}, x_i,…., x_n>) ∨

(LSt = <x_1, ..., x_i,…., x_n>, lst = <x_1, ..., x_i,…., x_n>

- **irA: Lista X Entero→Lista**

/* Coloca la ventana sobre el pos-ésimo elemento de la lista */

{ post: (pos < 1 ∨ pos > n, lst = < x_1, ..., x_n >) ∨ (lst = < x_1, ..., xpos , ... , x_n >) }

ALICIA GIOIA
CLAUDIA LITVAK

- **irPrimero: Lista →Lista**

/* Coloca la ventana sobre el primer elemento de la lista */

{ pre: lst = LST }

{ post: (LST = < >□, lst = < >□) ∨ (LST = < x_1, ..., x_n >, lst = < x1 , ..., x_n >) }

- **irUltimo: Lista →Lista**

/* Coloca la ventana sobre el último elemento de la lista */

{ pre: lst = LST }

{ post: (LST = < >□, lst = < >□) ∨ (LST = < x_1, ..., x_n >, lst =< x_1, ..., xn >) }

- **verActual: Lista → Elemento**

/* Retorna el elemento de la ventana */

{ pre: lst = < x_1, ..., xi , ... , x_n > }

{ post: infoLista = x_i }

- **verPosActual: Lista → Entero**

/* Retorna la posición del actual*/

(pre: lst = LST = <x_1, ..., x_i,…., x_n>)

(post: i

- **longitud: Lista → Entero**

/* Retorna el número de elementos de la lista */

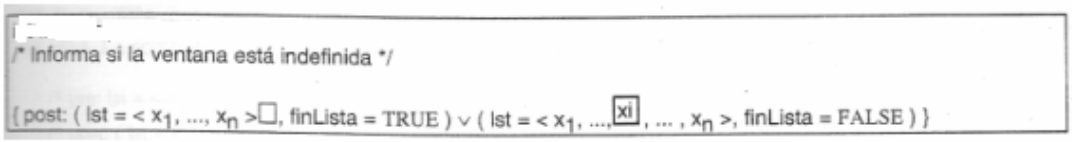
{ post: longLista = n }

- **esVacia: Lista → Boolean**

/* Retorna true si la lista es vacía; false si no lo es

- **finLista: Lista → Boolean**

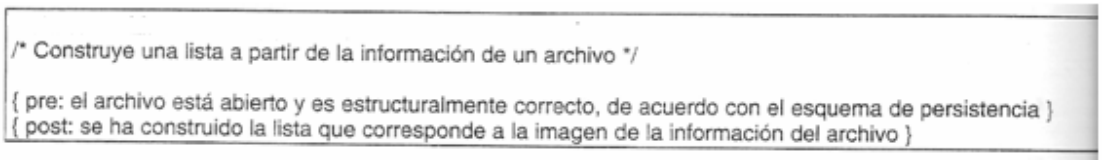
ALICIA GIOIA
CLAUDIA LITVAK



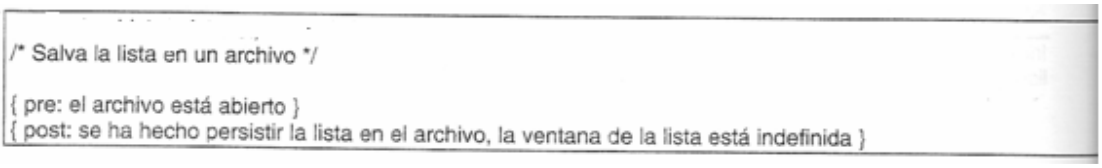
Otras operaciones interesantes:

El TAD definido anteriormente se puede enriquecer con operaciones de manejo de persistencia y destrucción, de acuerdo con la siguiente especificación:

cargarLista: Archivo → Lista



salvarLista: Lista → Archivo



Interface Lista

```
package Listas;
public interface Lista {
        void insertarDespues(Object x);
        void insertarAntes(Object x);
        void eliminar();
        void irSiguiente();
        void irAnterior();
        void irA (int n);
        Object verActual();
        int verPosActual();
        int longitud();
        boolean esVacio();
        boolean finLista();
}
```

Aplicaciones

La siguiente aplicación muestra el uso del TAD Lista. Para ello crea una lista con los enteros 1, 2, … , tam. También se han implementados los siguientes métodos, todos de complejidad O(n), donde n es la longitud de la lista.

1) Imprimir el contenido de una lista de enteros desde el primero al último.
2) Imprimir el contenido de una lista de enteros desde el último al primero.
3) Eliminar un elemento de la lista.
4) Construir una lista invertida a la dada.

ALICIA GIOIA
CLAUDIA LITVAK

5) Concatenar dos listas list1 y list2, dejando el resultado en la primera de ellas.

Observación: ***ListaE es una de las muchas clases que implementan el TAD Lista***

```
public class ListaAPP
{

  public Lista crearListaEnteros(int tam){
        Lista lst = new ListaE();
        for(int i = 1; i <= tam; i++)
           lst.insertarDespues(new Integer(i));
            return lst;
  }

  public void recorrer(Lista lst){
        lst.irPrimero();
        for (int i = 0; i < lst.longitud(); i ++){
           System.out.println(" " + lst.verActual());
           lst.irSiguiente();
        }
  }

  public void recorrerReves(Lista lst){
        lst.irUltimo();
        for (int i = 0; i < lst.longitud(); i ++){
           System.out.println(" " + lst.verActual());
           lst.irAnterior();
        }
  }

  public void eliminar(Lista lst, int n){
    Integer N = new Integer(n);
    lst.irPrimero();
    while((!lst.finLista()) && !(((Integer) lst.verActual()).equals(N)))
           lst.irSiguiente();
    if(((Integer)lst.verActual()) == n)
           lst.eliminar();
     else
           System.out.println( "Error");
     }

  public Lista invertirLista(Lista lst){
        Lista aux = new ListaE();
        lst.irPrimero();
        for(int cont = 1; cont <= lst.longitud(); cont++){
            aux.insertarAntes(lst.verActual());
            lst.irSiguiente();
        }
        return aux;
     }

   public void concatenar(Lista lst1, Lista lst2){
        lst1.irUltimo();
        lst2.irPrimero();
        for (int cont =0;cont <lst2.longitud() ; cont++){
            lst1.insertarDespues(lst2.verActual());
```

```
            lst2.irSiguiente();
        }
    }
}
```

```
class pruebaLista
{
      public static void main(String[] args) {
            ListaAPP lstAPP = new ListaAPP();

//Construye una lista con los enteros 1, 2, ... ,18
        Lista lst = lstAPP.crearListaEnteros(18);

//Recorre la lista desde el primer elemento al último
        lstAPP.recorrer(lst);

//Recorre la lista desde el último elemento al primero
        lstAPP.recorrerReves(lst);

//Elimina el numero 10 de la lista
        lstAPP.eliminar(lst,10);

//Invierte la lista
        Lista lst1 = lstAPP.invertirLista(lst);

//Concatenar dos listas
        lstAPP.concatenar(lst,lst1);
        lstAPP.recorrer(lst);
      }
}
```

Una aplicación: búsqueda

La operación ***búsqueda*** de un elemento en una lista recorre la lista desde el principio hasta encontrar el elemento o se llega al fin de la lista.

Normalmente cada elemento de un conjunto se identifica por una **clave** (*key*). La clave puede estar formada por uno o más atributos.

Ejemplos

- cada alumno de un curso se identifican por la matrícula
- cada auto de un conjunto de autos por la patente
- cada libro de una biblioteca por el ISBN

Vamos a considerar un conjunto de colectivos donde cada uno se identifica por el número de línea y el número de interno. Además cada colectivo tiene otros dos atributos que indican la cantidad de asientos y si es o no apto para discapacitados. La siguiente aplicación construye una lista de colectivos e implementa un método que dado un número de línea y un número de interno ***busca*** el colectivo en la lista de colectivos y muestra las características del mismo. Consideraremos que dos colectivos son iguales si coinciden la línea y el interno.

En primer lugar implementamos la siguiente interfaz.

```
public interface Comparable {
    int CompareTo(Comparable x); // se va a usar más adelante
    boolean equals(Comparable x);
}
```

Debemos definir nuestro propio *equals()* porque sólo necesitamos que sean igual la linea y el interno.

La clase colectivo debe implementar la interfaz anterior.

```
class Colectivo implements Comparable {
      private int linea;
      private int interno;
      private int cantAsientos;
      private boolean apto; //apto discapacitados

    public Colectivo(int linea, int interno){
      this.linea=linea;
      this.interno=interno;
      }

    public Colectivo(int linea, int interno, int cantAsientos, boolean
apto){
      this.linea=linea;
      this.interno=interno;
      this.cantAsientos = cantAsientos;
      this.apto = apto;
      }

    public void mostrar(){
      System.out.print("Linea: "+this.linea+" Nro interno: " +
                  this.interno + " Asientos: " + this.cantAsientos);
      if (this.apto)
           System.out.println(" ES apto");
      else
           System.out.println(" NO ES apto");
      }

    public boolean equals(Comparable x){
        Colectivo c = (Colectivo) x;
        return (this.linea == c.linea) && (this.interno == c.interno);
      }

    public int CompareTo(Comparable x){
           return 0;
      }
}

class ListaDeColectivos{
      private Lista lst;

      public ListaDeColectivos(){
            lst = new ListaE();
      }

      public void agregarColectivo(Colectivo c){
            lst.insertarDespues(c);
      }

    public void mostrar(int linea, int interno){
      Colectivo c = new Colectivo(linea, interno);
      lst.irPrimero();
      while(!lst.finLista()&&
```

```
                !((Colectivo) lst.verActual()).equals(c))
            lst.irSiguiente();
      if(lst.finLista())
            System.out.println("Colectivo no enconstrado");
      else
            ((Colectivo) lst.verActual()).mostrar();
      }
}

class  pruebaListaColectivos
{

    public static void main(String[] args)     {
       ListaDeColectivos colectivos  = new ListaDeColectivos();
       colectivos.agregarColectivo(new Colectivo(105,23,45,true));
       colectivos.agregarColectivo(new Colectivo(59,12,30,false));
       colectivos.agregarColectivo(new Colectivo(64,45,35,true));
       colectivos.mostrar(59,12);
       colectivos.mostrar(64,45);
       colectivos.mostrar(102,34);
       colectivos.mostrar(105,23);
   }
}
```

El algoritmo de búsqueda es claramente O(n)

```
lst.irPrimero();
while(!lst.finLista()&& !((Colectivo) lst.verActual()).equals(c))
   lst.irSiguiente();
   if(lst.finLista())
      System.out.println("Colectivo no enconstrado");
   else
      ((Colectivo) lst.verActual()).mostrar();
```

Ejercicios de utilización del TAD Lista

Parte I

1) Hacer una copia de una lista. La complejidad es O(n) , donde n es la longitud de la lista.
2) Localizar el elemento elem en la lista lst. Si hay varias ocurrencias del elemento, deja la ventana en la primera de ellas. La complejidad es O(n), donde n es la longitud de la lista.
3) Eliminar todas las ocurrencias de un elemento en una lista. La complejidad es O(n), donde n es la longitud de la lista.
4) Decidir si dos listas son iguales. La complejidad es O(n), donde n es el mayor valor entre las longitudes de las listas. En el peor de los casos las listas tienen la misma longitud, y debe recorrerlas hasta el final antes de informar que son iguales. Dicho valor se puede acotar siempre con la mayor de las longitudes de las listas.
5) Indicar si una lista de enteros se encuentra ordenada. La complejidad es O(n), donde n es la longitud de la lista.
6) Indicar si una lista es sublista de otra.

Parte 2

Especifique formalmente precondición y poscondición y desarrolle el codigo java para los siguientes problemas. Calcule la complejidad de la solucion.

1) void adicLista(Lista lista, Object elem)
   /* Adiciona el elemento elem al final de lista */
2) void sustLista(Lista lista, Object elem)

/* Sustituye el contenido actual de la ventana por el elemento elem */
3) boolean estaLista(Lista lista, Object elem)
/* Indica si el elemento elem aparece en la lista */
4) void simplificarLista(Lista lista)
/* Deja en la lista una sola ocurrencia de cada uno de los elementos presentes */
5) int numDiferentes(Lista lista)
/* Retorna el número total de elementos diferentes en lista */
6) int numOcurre(Lista lista, Object elem)
/* Calcula el número de veces que aparece elem en lista */
7) int ultOcurre(Lista lista, Object elem)
/* Retorna la posición de la última aparición de elem. Si no ocurre retorna 0 */

Parte 3
Ejercicio 1

La siguiente es una especificación de lista algo diferente a la dada en clase. Para cada una de las operaciones, L representa una lista concreta y también se asume que uno de los elementos es el actual.

1. **Insertar(L, x, i)**. Añade el elemento x a L en la posición i.
2. **Añadir (L, x)**. Añade el elemento al final de la lista L.
3. **Obtener(L,i)**. Devuelve el elemento que se encuentra en la posición i de L.
4. **Eliminar(L, i)**. Elimina el elemento que se encuentra en la posición i de L.
5. **Longitud(L)**. Devuelve la cantidad de elementos de L.
6. **Incicio(L)**. Sitúa la posición actual en el primer elemento de L.
7. **Actual(L)**. Devuelve la posición del actual.
8. **Siguiente(L)**. Incrementa la posición del actual.

a) Escribir dominio, codomino y las pre y postcondiciones para cada una de las operaciones para que éstas funciones correctamente.
b) Escribir la interface en Java.
c) Implementar estas operaciones usando las primitivas dadas en clase.

Nota: Esta especificación está dada en:
Estructuras de Datos, Algoritmos, y Programación Orientada a Objetos. Gregory Heileman. Mc. Graw Hill. 1997.

Ejercicio 2

Hay muchas maneras de especificar una lista. En este ejercicio vamos a considerar una completamente diferente: no hay elemento actual ni operaciones de insertar y eliminar en forma explícita. La llamaremos SLIST. Las operaciones vaciar y esVacia se definen como antes y además consta de las siguientes seis operaciones. Z es el conjunto de números enteros y L es el conjunto de valores de SLIST.

1. **Tamaño**: L -> Z: La función Tamaño(L) devuelve la cantidad de elementos de L
2. **Truncar**: Z X L -> L: La función Truncar(z, L) está indefinida si |z| es mayor que el tamaño de L. En otro caso, su valor es L', donde, si z es no negativo, entonces L' consiste de los primeros z elementos de L; en otro caso, L' consiste de los últimos –z elementos de L.
3. **Concatenar**: L X L -> L: La función Concatenar (L1, L2) es una lista L tal que Truncar(Tamaño(L1), L) = L1, Truncar(-Tamaño(L2),L) = L2, y Tamaño(L) = Tamaño(L1) + Tamaño(L2).
4. **ExaminarPrimero**: L -> E: La función ExaminarPrimero(L) es indefinida si L es vacía; en otro caso, devuelve el primer elemento de L.

5. **ExaminarUltimo**: L -> E: La función ExaminarUltimo(L) es indefinida si L es vacía; en otro caso, devuelve el último elemento de L.
6. **UnicoElem**: E -> L: La función UnicoElem(e) es una lista de tamaño 1 que consiste del único elemento e.

Notar que para una lista de tamaño 1, ExaminarPrimero y ExaminarUltimo retornan el mismo valor. Las constructoras son UnicoElem y Concatenar, la destructora es Truncar y las operaciones Tamaño, ExaminarPrimero y ExaminarUltimo son analizadoras.

Implementar, con estas operaciones primitivas, las siguientes operaciones:
i) insertarAntes ii) insetarDespues iii) eliminar iv) recorrer la lista del principio al final

Nota: Esta especificación está dada en:
Data Structures, Algorithms, and Performance. Derick Wood. Addison Wesley. 1993.

Ejercicio 3

Otra especificación del TDA Lista. La operación esVacia se define como en las otras especificaciones. Aquí tampoco hay elemento actual.
1. **Primero(L)**: Devuelve el valor del primer elemento de la lista.
2. **Insertar(L, x)**: Devuelve la lista resultante de poner a x como primer elemento de L.
3. **Resto(L)**: Devuelve la lista sin el primer elemento.
4. **Modificar(L, x)**: Cambia el primer elemento de L por x.

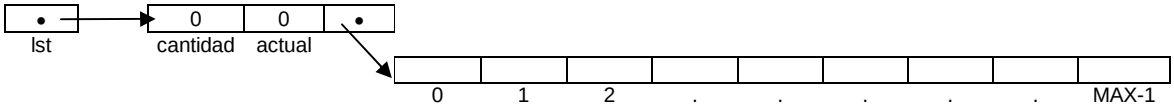
a) Escribir dominio, codomino y las pre y postcondiciones para cada una de las operaciones para que éstas funciones correctamente.
b) Demostrar que estas operaciones puede ser implementadas con las primitivas de Lista dadas en clase.
c) Implementar, en función de estas operaciones primitivas, las siguientes operaciones:
i) insertarAntes ii) insetarDespues iii) eliminar iv) recorrer la lista del principio al final

Implementaciones del TAD LISTA

Implementación estática:

En esta representación los elementos de la lista se almacenan consecutivamente en un arreglo de un tamaño máximo (MAX) predefinido. Además del arreglo, una lista tiene dos atributos adicionales de tipo entero que llamaremos ***cantidad*** y ***actual*** para indicar la cantidad de elementos de la lista y la posición del elemento actual. Si el valor de ***actual*** es menor que 1 o mayor que cantidad decimos que actual está indefinido.

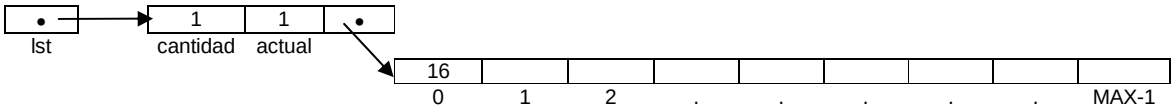
La lista vacia (lst = <>) se representa



Una lista con un único elemento siendo éste el actual.

lst = <**16**>

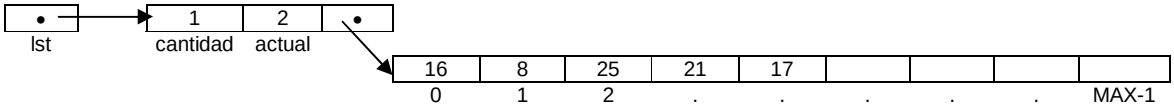
ALICIA GIOIA
CLAUDIA LITVAK



Notar que el elemento **actual** de la lista se almacena en la posición ***actual-1*** del arreglo.

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

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



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

Codificación en Java de estos métodos

```
public class ListaE implements Lista
{
   private int actual;
   private int cantidad;
   private Object dato[];
   private int defaultMax;

// ListaE crea una lista vacia
   public ListaE()      {
     defaultMax = 10;
     actual = 0;
     cantidad = 0;
     dato = new Object[defaultMax];
   }

//Modificadoras
    public void irSiguiente(){
        actual++;
    }

    public void irAnterior(){
         actual--;
    }

    public void irA(int pos){
        actual = pos;
    }
```

```
    public void irPrimero(){
        actual = 1;
    }

    public void irUltimo(){
        actual=cantidad;
    }

    //Analizadoras
    public Object verActual(){
        return dato[actual-1];
    }

    public int verPosActual(){
       return actual;
    }

    public int longitud(){
       return cantidad;
   }

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

   public boolean finLista(){
       return ((actual <1) || (actual>cantidad));
   }

}
```

Insertar después del actual

Si debemos insertar después del actual debemos correr todos los elementos siguientes una posición hacia la derecha, poner el nuevo elemento en la posición actual y aumentar los valores de cantidad y actual en 1. Puede ocurrir que ya no haya espacio en el arreglo para realizar la inserción en cuyo caso debemos agrandar el arreglo, por ejemplo duplicarlo.

```
   public void insertarDespues(Object x){
       if (cantidad == defaultMax)
           duplicarArreglo();
       for (int pos = cantidad-1; pos > actual-1; pos--)
               dato[pos+1] = dato[pos];
       dato[actual]=x;
       actual++;
       cantidad++;
   }


   private void duplicarArreglo(){
       defaultMax = defaultMax*2;
       Object[] nuevo = new Object[defaultMax];
       for (int pos = 0; pos < cantidad; pos++)
            nuevo[pos] = dato[pos];
       dato=nuevo;
   }
```

Insertar antes del actual

Si debemos insertar antes del actual debemos correr el actual y todos los siguientes una posición hacia la derecha, poner el nuevo elemento en la posición actual-1 y aumentar cantidad en 1. En este caso tenemos que considerar el caso especial de lista vacía.

```
public void insertarAntes(Object x){
     if (cantidad == defaultMax)
         duplicarArreglo();
     if (cantidad!=0    ){
         for (int pos = cantidad-1; pos >= actual-1; pos--)
             dato[pos+1] = dato[pos];
         dato[actual-1]=x;
     }
     else {
         dato[0] = x;
         actual = 1;
     }
     cantidad++;
  }
```

Eliminar el actual
Si debemos eliminar el actual hay que correr los elementos siguientes una posición adelante y disminuir cantidad en 1.

```
   public void eliminar(){
      for(int pos = actual-1; pos < cantidad-1; pos++)
          dato[pos] = dato[pos+1];
      cantidad--;

   }
```

Estos tres métodos son O(n). Al insertar un nuevo elemento puede ocurrir que ya no haya lugar en el arreglo, en tal caso debemos agrandar el arreglo. Este es uno de los puntos más débiles de esta representación. Supongamos que definió a MAX como 10 al agregar un nuevo elemento duplicamos el arreglo con lo que estamos trabajando con un arreglo de tamaño 20. Si luego hacemos cinco eliminaciones el tamaño de la lista es 6 pero el arreglo tiene tamaño 20, con el consiguiente desperdicio de memoria. Esta representación es adecuada sólo en el caso en que la lista sea de tamaño casi constante y no haya muchas inserciones y eliminaciones a lo largo de su ciclo de vida, por ejemplo la lista de alumnos de un curso.

Implementación dinámica:
Existen varias maneras de representar el TAD Lista en forma dinámica. Analizaremos algunas de ellas.