

Estructuras Estáticas vs. Estructuras Dinámicas

Las estructuras vistas hasta ahora: ARREGLOS

Los arreglos son, probablemente, la estructura más usada para almacenar y ordenar datos dentro de la programación, debido a que la sintaxis para acceder a sus datos es muy amigable para el programador.

Por ejemplo, si tenemos un arreglo de int's, y queremos sumar el segundo número de un arreglo con el cuarto, simplemente usamos la notación [] para acceder a ellos.

Estructuras Estáticas vs. Estructuras Dinámicas

Desventajas de esta forma de almacenamiento:

1) Los arreglos tienen una capacidad de almacenamiento determinada y no modificable: problema si queremos almacenar datos sin saber cuantos almacenaremos en total, o si la cantidad de datos es variable.

1) Por (1), los programadores utilizan arreglos “lo suficientemente grandes”, desperdiciando espacio de memoria que nunca se utiliza.

Ejemplo: si queremos registrar a los clientes que compraron en una tienda cada día, podríamos crear un arreglo de una capacidad estimada de 100 espacios para cada día. Pero en el caso que en una ocasión sólo entraran a comprar 22 clientes, 78 espacios del arreglo son desperdiciados. Peor aún, si entraran 150 personas a comprar y el arreglo no fuera capaz de almacenar a todos los clientes.

Estructuras Estáticas vs. Estructuras Dinámicas

Desventajas de esta forma de almacenamiento:

- La inserción de un objeto al principio del arreglo, sin sobrescribir el primer espacio, se torna mas complicada, pues debemos correr en un espacio a la derecha a todo el resto de los datos.
- Los arreglos no manejan los datos de forma dinámica.

Para solucionar este problema, haremos uso de una nueva forma de almacenamientos:

las estructuras datos **dinámicas**, con su principal estructura:
la Lista Enlazada.

Lista Enlazada simple

Qué es una **lista enlazada**?

La Lista Enlazada Simple es la estructura de datos dinámica fundamental basada en punteros. Del concepto fundamental de ella derivan las otras estructuras de datos dinámicas.

Para solucionar un problema como el presentado anteriormente, necesitamos una estructura que sea capaz de modificar su capacidad, es decir, que maneje los datos de forma dinámica.

Lista Enlazada simple

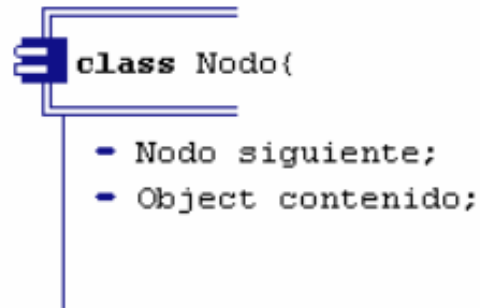
La memoria:

- Un arreglo asigna memoria para todos sus elementos ordenados como **un sólo bloque**.
- La lista enlazada asigna espacio **para cada elemento** por separado, en su propio bloque de memoria, llamado **nodo**. La lista conecta estos nodos usando **punteros**, formando una estructura parecida a la de una cadena.

Lista Enlazada simple

Un **nodo** es un **objeto** como cualquier otro, y sus atributos serán los encargados de hacer el trabajo de almacenar y **apuntar** a otro nodo.

Cada nodo tiene dos atributos: un atributo “contenido”, usado para almacenar un objeto; y otro atributo “siguiente”, usado para hacer referencia al siguiente nodo de la lista.

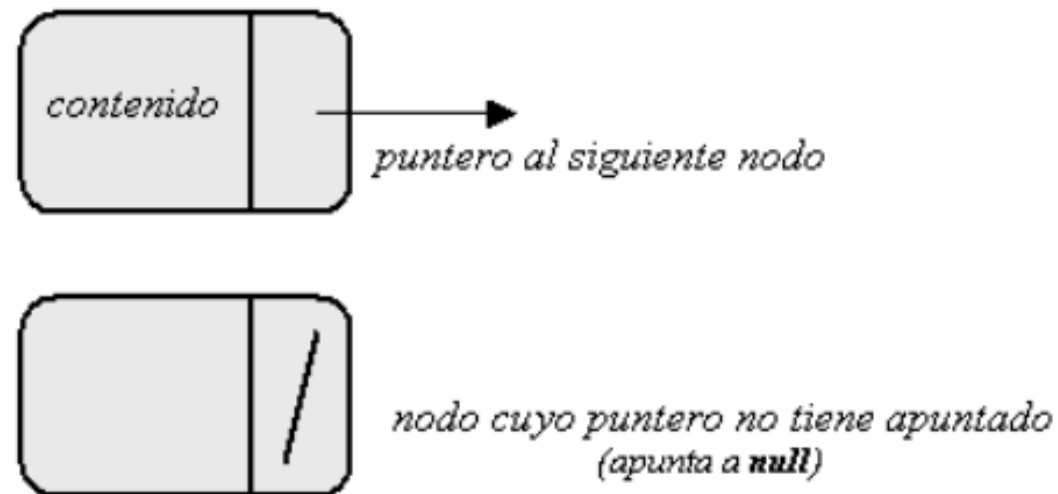


```
classDiagram
    class Nodo {
        - Nodo siguiente;
        - Object contenido;
    }
```

The diagram shows a class named `Nodo` with two attributes: `- Nodo siguiente;` and `- Object contenido;`. The class is represented by a rectangle with a small tab on the left side.

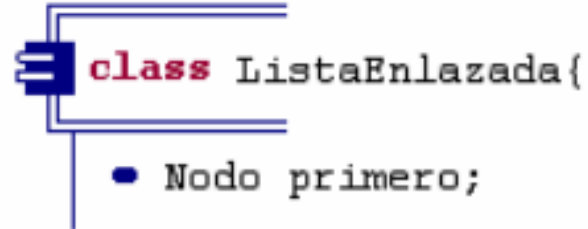
Lista Enlazada simple

Los nodos serán representados por los siguientes símbolos:



Lista Enlazada simple

La parte frontal de la lista es representada por un puntero al primer nodo. Es decir, un atributo de la lista enlazada es un nodo **primero**.

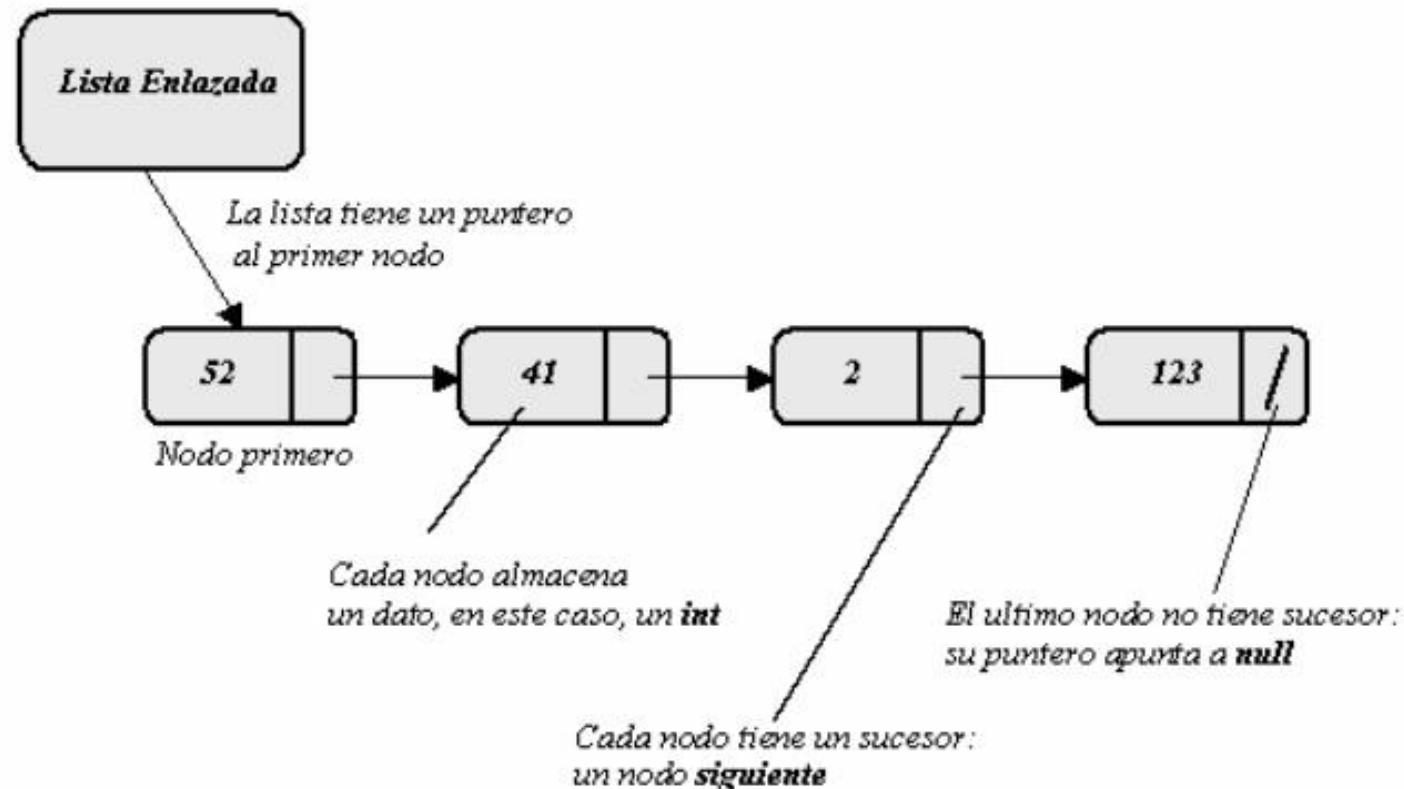


```
classDiagram
    class ListaEnlazada {
        +Nodo primero
    }
```

The diagram shows a class named `ListaEnlazada` with a single attribute `Nodo primero`. The class is represented by a rectangle with a blue border and a blue header bar. The attribute is shown as a bullet point inside the class box.

Lista Enlazada simple

Podríamos representar a una lista enlazada que almacena enteros de la siguiente manera:



Lista Enlazada simple

La lista también tendrá un atributo largo, del tipo **long** (o int) que representará la cantidad de elementos en ella.

```
class ListaEnlazada{  
    • Nodo primero;  
    • long largo;  
}
```

Lista Enlazada simple

Cómo acceder a un elemento de la lista?

Para tener acceso a un elemento de la lista, hay que hacer uso de los punteros. Se crea un puntero al primer nodo de la lista y se avanza por medio del atributo “siguiente”.

Ejemplo: lista enlazada llamada **lista** (tiene almacenado nombres de personas por medio de String's).

Si quisiéramos tener acceso al cuarto nombre almacenado, tendríamos que:

1- Crear un puntero al primero de la lista

```
// creamos un puntero al primer nodo de la lista  
-> Nodo puntero=lista.primer;
```

Lista Enlazada simple

Cómo acceder a un elemento de la lista?

Ejemplo: lista enlazada llamada **lista** (tiene almacenado nombres de personas por medio de String's).

2- Avanzar con el puntero hasta el cuarto nodo

3- Utilizar el contenido del nodo

```
// lo hacemos avanzar hasta el 4 nodo
- int i=1;
- while (i<4) {
    - puntero=puntero.siguiente;
    - i++;
}

// usamos el contenido del nodo para algo útil
- System.out.println("El 4to nombre almacenado es "+puntero.contenido);
```

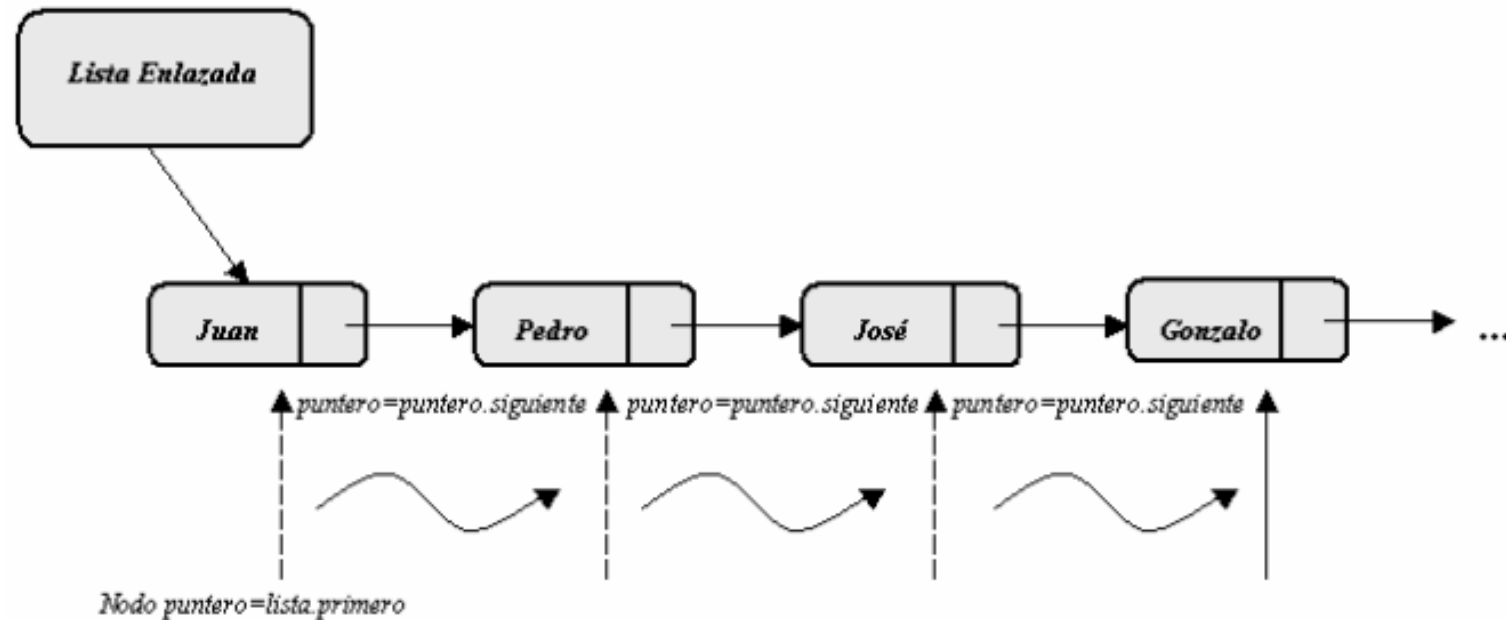
Lista Enlazada simple

Cómo acceder a un elemento de la lista?

```
// creamos un puntero al primer nodo de la lista  
-● Nodo puntero=lista.primeronodo;  
  
// lo hacemos avanzar hasta el 4 nodo  
-● int i=1;  
-● while(i<4) {  
-   puntero=puntero.siguiente;  
-   i++;  
- }  
  
// usamos el contenido del nodo para algo útil  
- System.out.println("El 4to nombre almacenado es "+puntero.contenido);
```

Lista Enlazada simple

Acceso a los elementos de la lista



Lista Enlazada simple. Ejemplo.

El nombre de la clase será **ListaEnlazada**.

- La lista almacenará sólo un tipo de datos, que serán objetos de la clase **Cliente**, (definida por el programador).
- Es necesario que los Clientes tengan un atributo único para cada uno, de tal forma que podamos diferenciarlos. Este atributo será el **RUT** (dni en nuestro dominio).

Lista Enlazada simple. Métodos

```
public class
Cliente{

    - String rut;
    - String nombre;
    - String apellido;
    - boolean esClienteFrecuente;

    public
    Cliente(String rut, String nombre, String apellido, boolean esClienteFrecuente){

        this.rut=rut;
        this.nombre=nombre;
        this.apellido=apellido;
        this.esClienteFrecuente=esClienteFrecuente;
    }

}
```


Lista Enlazada simple. Métodos

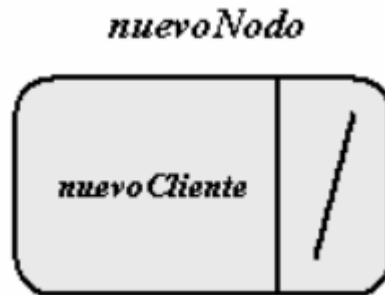
Construimos ahora la clase ListaEnlazada. En esta implementación de lista enlazada, definiremos los siguientes métodos:

NOMBRE DEL METODO	VALOR DE RETORNO	TIPOS DE ARGUMENTO	UTILIDAD
estaVacia()	boolean	Ninguno	Retorna true si la lista está vacía, sino, false
vaciar()	void	Ninguno	Remueve todo el contenido de la lista
largo()	long	Ninguno	Retorna el largo (numero de elementos) de la lista
tieneElRut(rut)	boolean	String	Retorna true si la lista tiene el rut especificado, sino, false
tengaMasElementos(puntero)	boolean	Nodo	Retorna true si puntero tiene sucesor, sino, false
añadir(nuevoCliente)	void	String	Añade a nuevoCliente al final de la lista
buscar(rut)	Cliente	String	Retorna el Cliente con el rut especificado
remover(rut)	void	String	Remueve al Cliente con el rut especificado. Devuelve true si elimina a alguno, sino, false

Lista Enlazada simple. Métodos

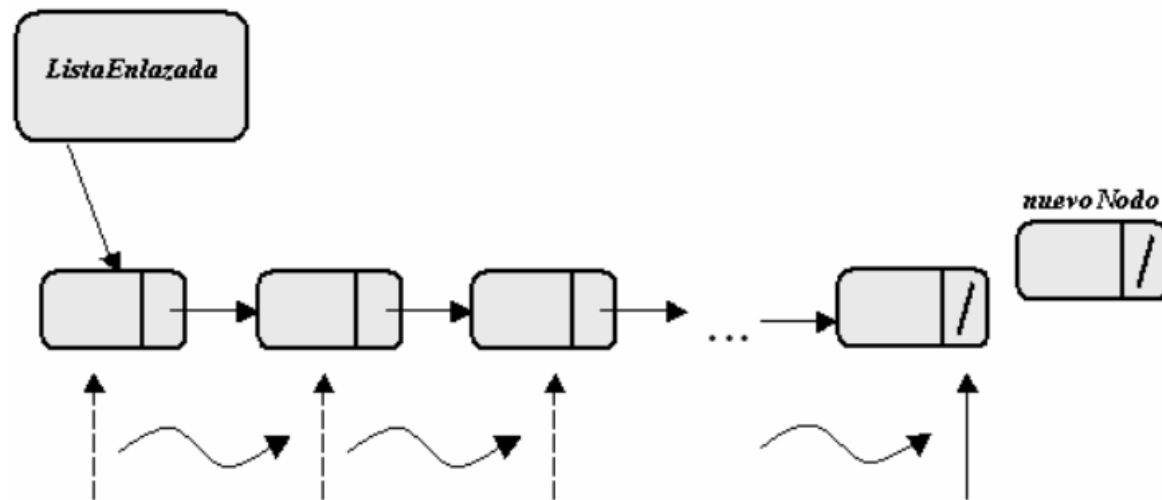
Método **añadir()**:

Para añadir un nodo a la lista, primero creamos un nodo (**nuevoNodo**), que contiene al **Ciente** pasado como argumento, al que llamaremos **nuevoCliente**.



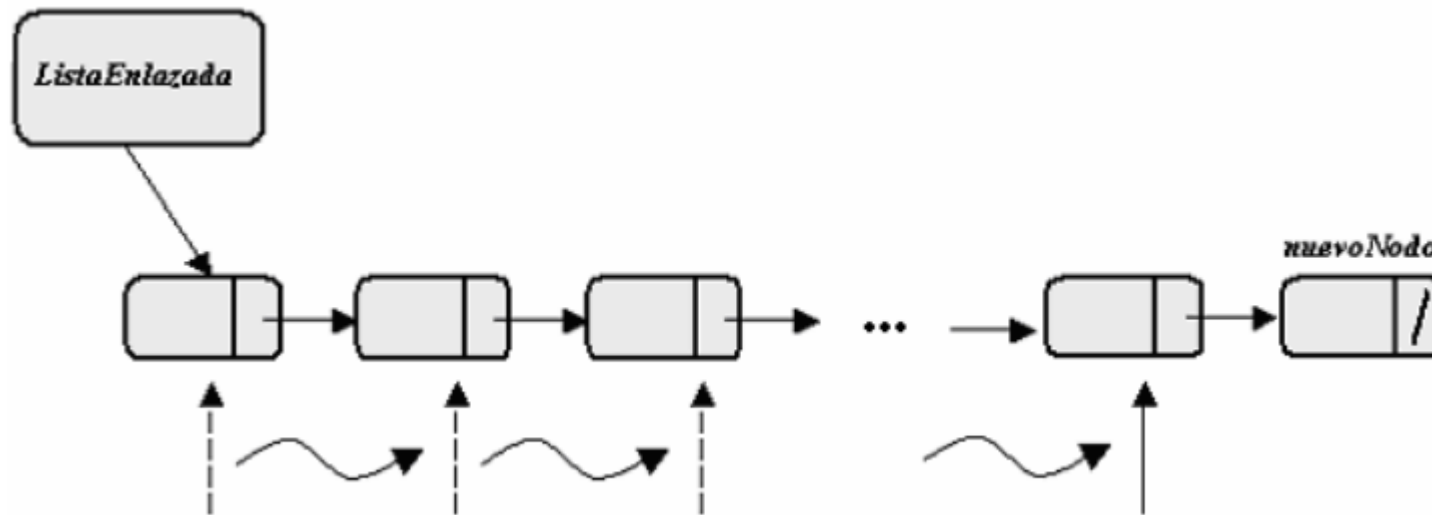
Lista Enlazada simple. Métodos

Luego conectamos a **nuevoNodo** con el último nodo de la lista. Para referirnos a éste, creamos un puntero, al que llamaremos **puntero**, al primer nodo y avanzamos hasta el último por medio de la instrucción `puntero = puntero.siguiente`



Lista Enlazada simple. Métodos

Puntero apunta al último nodo de la lista, y, además, ahora puntero.siguiente tiene un valor igual a null,
puntero.siguiente toma el valor nuevoNodo, es decir:
puntero.siguiente = nuevoNodo .



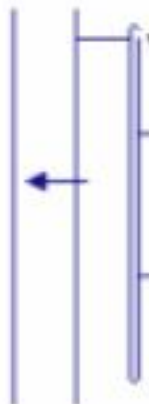
Lista Enlazada simple. Métodos

Método **buscar()**

Para encontrar un Cliente en la lista, debemos buscarlo por medio de su atributo único: el RUT para este ejemplo.

Crearemos un puntero **puntero** con el que recorreremos la lista, comparando el RUT especificado con el RUT de cada Cliente, hasta encontrar una coincidencia. Para ésto, tendremos que usar una instrucción:

```
while (puntero != null) { // mientras el puntero no apunte al vacío
    if (puntero.contenido.rut.compareTo(rut) == 0) // si los RUTs son iguales...
        return puntero.contenido; // ...entonces retorna el contenido del nodo apuntado
    puntero = puntero.siguiente; // en otro caso, se avanza al siguiente hasta encontrar
                                // una coincidencia
}
```

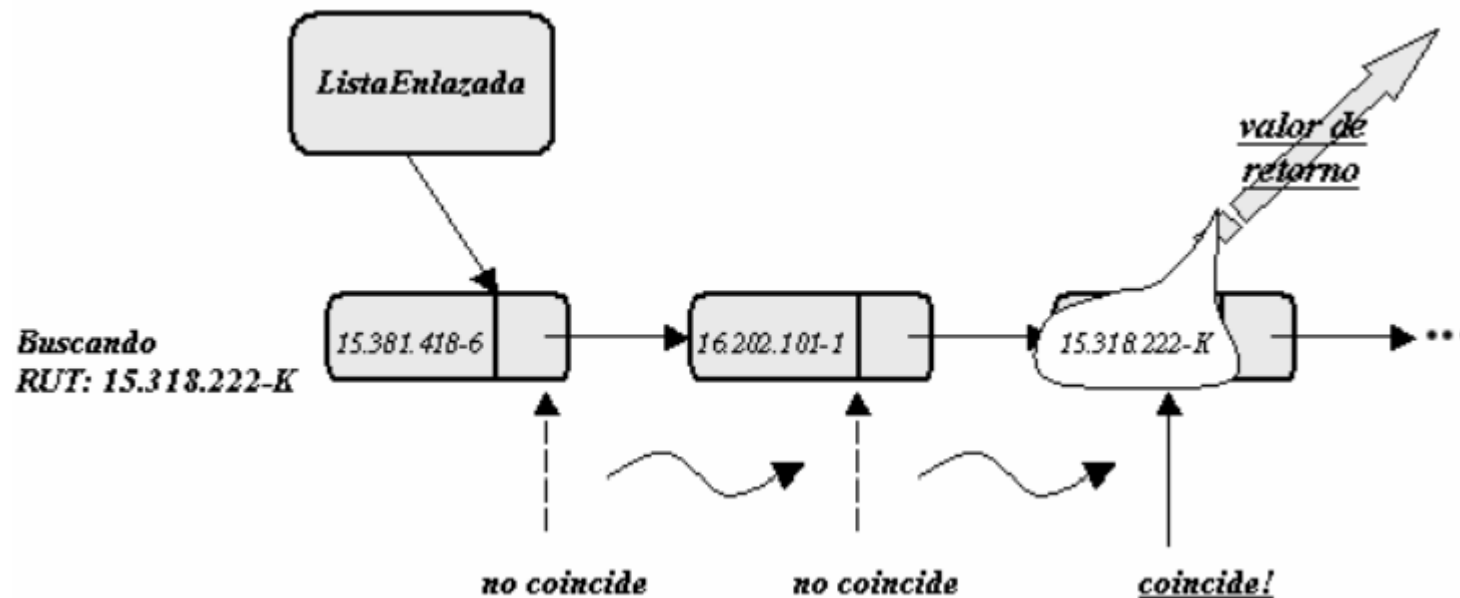


```
class Nodo {
    - Nodo siguiente;
    - Object contenido;
}
```

Lista Enlazada simple. Métodos

Método **buscar()**

Para encontrar un Cliente en la lista, debemos buscarlo por medio de su atributo único: el RUT.



Lista Enlazada simple. Métodos

Método **remove()**:

Para eliminar un nodo de la lista, primero debemos ubicar el nodo anterior éste, es decir, **puntero.siguiente** será el nodo a eliminar.

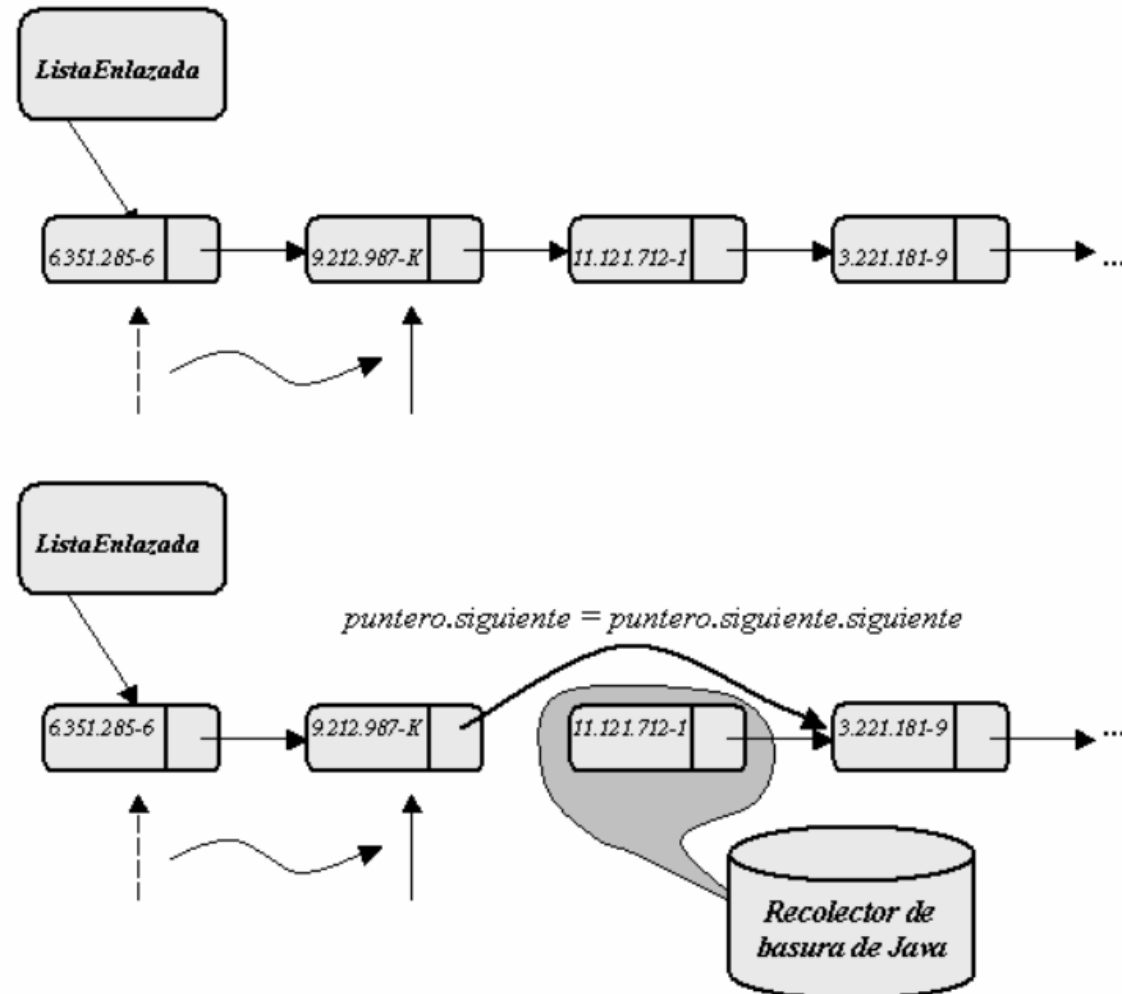
Ahora, simplemente dejamos sin puntero al nodo:

```
puntero.siguiente = puntero.siguiente.siguiente  
el Recolector de Basura de Java lo recoge y se elimina
```

Lista Enlazada simple. Métodos

Método **remove()**:

*Eliminar
RUT: 11.121.712-1*



Lista Enlazada simple. Resumen

- Las estructuras enlazadas se forman enlazando nodos. Los nodos son las células con las que construimos la estructura.
- Los nodos contienen vínculos a otros nodos, lo que permite construir la estructura.
- Nos se tiene acceso directo a cada nodo de la estructura, implica que se debe recorrer la estructura cuando buscamos algún nodo en particular.
- Puede recorrerse una estructura enlazada de dos maneras diferentes:
 - El nodo auxiliar “mira” el nodo sobre el que está posado. Este tipo de recorrido se usa, por ejemplo, para recuperar o actualizar un valor.
 - El nodo auxiliar “mira” al nodo siguiente del nodo sobre el que está posado. Este tipo de recorrido se usa, por ejemplo, para insertar o eliminar un nodo. En este caso, el nodo inicial debe ser tratado separadamente,

Lista Enlazada simple. Resumen

- Para eliminar un nodo, se lo circunvala, apuntando su anterior a su siguiente.
- Un elemento eliminado sigue existiendo en la memoria; sólo se ha vuelto inaccesible.
- Si se pierde la referencia a una estructura, se pierde la estructura, pues ésta se vuelve inaccesible.
- Por esta razón, se usa un nodo auxiliar que es una copia de la referencia del origen de la lista y no esta referencia directamente.
- En **teoría**, una estructura enlazada tiene una capacidad ilimitada.

TDA PILA. Implementación con EDD dinámica

La interfaz de *PilaTDA*

1. `public interface PilaTDA; {`
2. `void InicializarPila();` // sin precondiciones
3. `void Apilar(int x);` // pila inicializada
4. `void Desapilar();` // pila inicializada y no vacía
5. `boolean PilaVacía();` // pila inicializada
6. `int Tope();` // pila inicializada y no vacía
7. `}`

TDA PILA. Implementación con EDD dinámica

Los datos se representarán como **nodos** en una **lista enlazada**.

Cada **nodo** corresponderá a un **elemento que se apila**. La estructura contendrá, por lo tanto, tantos nodos como elementos contenga la pila representada.

La **referencia** de la estructura enlazada será el **primer nodo**, que llamaremos **primero**. Los nodos se **agregan al principio de la lista**.

Luego, cada operación Apilar o Desapilar cambiará el **origen** de la estructura enlazada.

TDA PILA. Implementación con EDD dinámica

```
public class PilaLD implements PilaTDA {  
    class Nodo {                                // la célula de la estructura  
        int info;                               // el valor almacenado  
        Nodo sig;                              // la referencia al siguiente nodo  
    }  
    Nodo primero;                              // la referencia a la estructura  
    public void InicializarPila() {  
        primero = null;  
    }  
    public void Apilar (int x) {                // el nuevo elemento se agrega al inicio  
        Nodo nuevo = new Nodo();  
        nuevo.info = x;  
        nuevo.sig = primero;  
        primero = nuevo;                       // nueva referencia a la estructura  
    }  
    public void Desapilar() {  
        primero = primero.sig;                 // nueva referencia a la estructura  
    }  
    public boolean PilaVacía() {  
        return (primero == null);  
    }  
    public int Tope() {  
        return (primero.info);  
    }  
}
```

TDA PILA. Análisis de costos

Costo computacional, métodos EDD Estática (Arreglo)

El método **InicializarPila** tiene complejidad constante

El método **Apilar** tiene complejidad lineal. Esto se debe al corrimiento, que requiere tantos desplazamientos de elementos individuales como elementos tenga la pila.

```
public void Apilar(int x) {  
    for (int i = inx-1; i >= 0; i--) {  
        arr[i+1] = arr[i];  
    }  
    arr[0] = x;  
    inx++;  
}
```

Esto es porque el ciclo de las líneas 2 a 4 repite una operación de costo constante tantas veces como elementos tenga la pila.

TDA PILA. Análisis de costos

Costo computacional, métodos EDD Estática (Arreglo)

El método **Desapilar** también tiene **costo lineal** por las mismas razones:

```
public void Desapilar() {  
    for (int i = 0; i < inx; i++) {  
        arr[i] = arr[i+1];  
    }  
    inx --;  
}
```

Los métodos **PilaVacía** y **Tope** tienen ambos costo constante

TDA PILA. Análisis de costos

Costo computacional, métodos EDD Dinámica (Listas enlazadas)

Recordemos que el tamaño del problema es la cantidad de elementos en la pila.

El método **InicializarPila** tiene costo constante:

```
public void InicializarPila() {  
    primero = null;  
}
```

Aquí sólo tenemos una asignación de valor. El método **Apilar** tiene costo constante:

```
public void Apilar (int x) {  
    Nodo nuevo = new Nodo();  
    nuevo.info = x;  
    nuevo.sig = primero;  
    primero = nuevo;  
}
```

Todas operaciones de costo constante que no dependen de la cantidad de elementos de la pila

TDA PILA. Análisis de costos

Trabajo Práctico TP4:

Completar el análisis de los costos en ambas implementaciones