Programación II – ALGORTIRMOS Y ESTRUCTURAS DE DATOS II

**Docente:** Esp. Lic. PEREZ, Nicolás Ignacio

[nicoperez@uade.edu.ar](mailto:nicoperez@uade.edu.ar)

Repositorio de la materia: <https://github.com/NicolasPerezUNLaSMN/PROG_II_UADE_JAVA>

# Facultad de Ingeniería y Ciencias Exactas

Universidad Argentina de la Empresa



**Explicación de Listas Dinámicas con el Paradigma de TDA en Java**

### ¿Qué es una Lista Dinámica?

Una lista dinámica es una estructura de datos que permite almacenar una secuencia de elementos de manera flexible, donde el tamaño puede crecer o reducirse durante la ejecución del programa. A diferencia de los arreglos tradicionales, no es necesario definir un tamaño fijo al momento de declararla.

En Java, se puede implementar una lista dinámica utilizando nodos enlazados, formando lo que se conoce como una **lista simplemente enlazada**.

### ¿Qué es un TDA (Tipo de Dato Abstracto)?

Un TDA define una estructura de datos por su comportamiento (operaciones disponibles), sin especificar cómo está implementado internamente. En Java, esto se logra utilizando **interfaces**.

Usar interfaces permite:

* Separar la definición de la estructura del cómo se implementa.
* Facilitar el mantenimiento y la extensión del código.
* Promover el principio de programar "hacia una interfaz".

### TDA para Lista Dinámica

Para modelar una lista dinámica como TDA, se crean dos interfaces:

* INodo: Representa un nodo de la lista. Define métodos para acceder y modificar el dato y el siguiente nodo.
* ILista: Define las operaciones que se pueden realizar sobre la lista (insertar, eliminar, buscar, etc).

Un reloj con números romanos

El contenido generado por IA puede ser incorrecto.

A continuación se detalla la interfaz ILista con sus primitivas, condiciones de uso (precondiciones), efectos esperados (poscondiciones) y axiomas:

public interface ILista {

// Pre: -

// Pos: Retorna true si la lista no contiene elementos

// Ax: esVacia() <=> cantidadElementos() == 0

boolean esVacia();

// Pre: dato definido

// Pos: Inserta el dato al inicio de la lista

void insertarPrimero(int dato);

// Pre: dato definido

// Pos: Inserta el dato al final de la lista

void insertarUltimo(int dato);

// Pre: posicion >= 0 y <= cantidadElementos(), dato definido

// Pos: Inserta el dato en la posición indicada

void insertarPosicion(int dato, int posicion);

// Pre: La lista no debe estar vacía

// Pos: Elimina el primer elemento de la lista

void eliminarPrimero();

// Pre: La lista no debe estar vacía

// Pos: Elimina el último elemento de la lista

void eliminarUltimo();

// Pre: posicion >= 0 y < cantidadElementos()

// Pos: Elimina el elemento en la posición indicada

void eliminarPosicion(int posicion);

// Pre: La lista no debe estar vacía

// Pos: Retorna el primer elemento de la lista

int obtenerPrimero();

// Pre: La lista no debe estar vacía

// Pos: Retorna el último elemento de la lista

int obtenerUltimo();

// Pre: posicion >= 0 y < cantidadElementos()

// Pos: Retorna el elemento en la posición indicada

int obtenerPosicion(int posicion);

// Pre: -

// Pos: Retorna la cantidad de elementos en la lista

int cantidadElementos();

// Pre: dato definido

// Pos: Devuelve la posición del dato si existe, -1 si no se encuentra

int buscarSecuencial(int dato);

// Pre: -

// Pos: Ordena los elementos de la lista en forma ascendente

void ordenarLista();

// Pre: -

// Pos: Muestra por consola los elementos de la lista en orden

void mostrarLista();

}

Estas primitivas definen completamente el comportamiento esperado de la lista dinámica, siguiendo el enfoque de TDA.

### Interfaces vs. Implementaciones

Cuando se usa una interfaz como tipo de dato (por ejemplo, INodo primero), se está indicando que ese atributo puede contener **cualquier objeto que implemente esa interfaz**, no necesariamente una clase específica.

#### Ventajas de usar interfaces como tipo:

* **Flexibilidad**: Se pueden cambiar las implementaciones sin afectar el resto del código.
* **Polimorfismo**: Se pueden utilizar distintas clases que implementan la misma interfaz, adaptándose a diferentes necesidades.
* **Bajo acoplamiento**: El código depende de abstracciones y no de detalles concretos.

#### Comparación:

* INodo primero; permite trabajar con cualquier implementación de nodo.
* Nodo primero; obliga a usar siempre la clase Nodo, limitando la extensibilidad.

### Aplicación del Paradigma TDA en Listas

Al aplicar el paradigma TDA:

* Se define **qué operaciones** debe tener la lista, sin preocuparse por **cómo** se implementan.
* La implementación concreta puede evolucionar o cambiar (por ejemplo, usar nodos con información adicional, como etiquetas o prioridades), sin modificar el código que utiliza la lista.

Esto permite una programación más ordenada, modular, escalable y mantenible.

### Conclusión

Utilizar el paradigma de TDA en la implementación de listas dinámicas en Java es una práctica recomendada que fomenta la abstracción, el bajo acoplamiento y la reutilización de código. Separar la interfaz de la implementación permite mayor libertad para mejorar o reemplazar partes del sistema sin afectar su funcionamiento general.

Este enfoque es especialmente útil en contextos educativos y profesionales donde se busca diseñar software robusto, limpio y mantenible.

### ¿Por qué esto es correcto?

java

CopiarEditar

ILista lista = new Lista();

Acá estás diciendo:

“Estoy creando una instancia de Lista, pero la voy a manipular **como si fuera una ILista**, es decir, solo voy a usar los métodos definidos en la interfaz.”

Esto es lo que se llama **programar hacia una interfaz**. Es uno de los principios más importantes de la **programación orientada a objetos**.

### Ventajas de hacer esto

1. **Abstracción**: No te importa cómo está hecha la lista, solo que cumpla lo que ILista promete.
2. **Flexibilidad**: Si mañana querés cambiar Lista por ListaCircular o ListaOrdenada (que también implementan ILista), ¡no tenés que tocar el resto del código!
3. **Polimorfismo**: Podés tener distintas clases que implementan ILista, y usarlas todas a través de esa misma referencia.

### ¿Cuándo **no** sería suficiente?

Si querés usar métodos **que estén en la clase Lista pero no en la interfaz ILista**, entonces sí necesitarías una referencia de tipo Lista. Pero si programaste bien tu interfaz, **eso no debería pasar**.

***Para practicar:***

* **Crear más primitivas en la lista:**

duplicarLista()  
busquedaBinariaEnLista()  
insertarEnOrden()

sonListasIguales()

invertirLista()

* **Modificar el Nodo y la Lista para hacer un Listado de Personas(dni y nombre), y probarlo en el main.**

**Trabajo Práctico 1**

El mismo se resuelve en grupos. Recuerden estar con grupo asignado en el Excel:

<https://docs.google.com/spreadsheets/d/11HrjVrljUUwaN4DXZxyK56ztzM1Cc7sAeIcvge8c-y8/edit?gid=290103117#gid=290103117>

El TP no es obligatorio, pero tiene nota, que puede favorecer el cierre de la cursada, además es fuente de práctica para el trabajo final y para el parcial.

En la clase trabajamos con lo que se llama una Lista Simplemente Enlazada, en el TP se pide crear el Nodo y la Lista**, doblemente enlazada**, es decir, es lo mismo pero cada Nodo apunta el siguiente y al anterior (ver imagen).

* Crear, modificar o investigar los archivos INodo, Nodo, ILista y Lista, hacer las mismas primitivas que teníamos con la lista simplemente enlazada.
* Una vez creado eso, hacer que una **Persona, tenga una Lista de Vehículos** (usando doblemente enlazada), probar las funciones básicas en el Test.

El trabajo debe estar subido en alguno de los repositorios de los integrantes de su equipo, una vez terminado dejarme el link del repositorio en la columna D del excel correspondiente a su equipo.

Estructura lista doblemente enlazada:

Imagen que contiene Escala de tiempo

El contenido generado por IA puede ser incorrecto.