

# Estructuras de datos lineales

- ▶ Materia Algoritmos y Estructuras de Datos - CB100
- ▶ Cátedra Schmidt
- ▶ Templates y Estructuras de datos dinámicas

# Templates en Java

El template es un mecanismo de abstracción por parametrización que ignora los detalles de tipo que diferencian a las funciones y TDAs.

El template indica que lo que se escribe es una plantilla. Cuando, en tiempo de compilación se genera código a partir de esa plantilla, se habla de una versión de la plantilla o template en el tipo especificado.

El compilador instancia o versiona la plantilla y realiza la correspondiente llamada cuando corresponda.

Al usar una plantilla, los tipos se deducen de los argumentos usados.

En Java, un **template** (también conocido como *plantilla* o **genérico**) es un mecanismo que permite crear clases, interfaces y métodos que puedan operar con tipos de datos de forma flexible, sin tener que definir el tipo de dato concreto en el momento de escribir el código. Esto se logra utilizando **generics** (genéricos).

## ¿Cómo funciona?

Los genéricos en Java te permiten escribir clases y métodos que son parametrizables en cuanto a los tipos de datos que manejan. Esto significa que puedes definir una clase o un método sin especificar el tipo exacto que usará, y luego, cuando lo utilices, especificas el tipo concreto.

```
public class Caja<T> {  
    private T contenido;  
  
    public void setContenido(T contenido) {  
        this.contenido = contenido;  
    }  
  
    public T getContenido() {  
        return contenido;  
    }  
}
```

En este caso, T es un tipo genérico. Cuando crees una instancia de Caja, puedes especificar el tipo de T:

```
Caja<String> cajaDeTexto = new Caja<>();
cajaDeTexto.setContenido("Hola");
System.out.println(cajaDeTexto.getContenido()); // Imprime: Hola

Caja<Integer> cajaDeNumero = new Caja<>();
cajaDeNumero.setContenido(123);
System.out.println(cajaDeNumero.getContenido()); // Imprime: 123
```

## Beneficios de los templates (genéricos)

Reutilización de código: Puedes crear clases o métodos que funcionen con diferentes tipos de datos, sin duplicar código.

Seguridad de tipos: Al definir el tipo de dato al instanciar, el compilador puede verificar los tipos en tiempo de compilación, evitando errores de tipo.

Legibilidad: Los genéricos ayudan a que el código sea más claro, al eliminar la necesidad de hacer casts entre tipos.

## Ejemplo de método genérico

También puedes hacer métodos genéricos dentro de clases que no son genéricas:

```
public class Util {  
    public static <T> void imprimir(T dato) {  
        System.out.println(dato);  
    }  
}  
  
Util.imprimir("Hola"); // Imprime: Hola  
Util.imprimir(42);     // Imprime: 42
```

En resumen, los templates o genéricos en Java son una herramienta poderosa que mejora la flexibilidad y reutilización de las clases y métodos, además de proporcionar seguridad en el manejo de tipos.

## Plantilla con mas de un tipo de dato

En Java, también puedes usar dos o más datos genéricos en una clase, método o interfaz. Para ello, simplemente defines múltiples parámetros de tipo genérico.

### Ejemplo de clase con dos datos genéricos

Aquí tienes una clase Par que puede almacenar dos tipos de datos genéricos distintos:

#### Main

```
Par<String, Integer> par = new Par<>("Edad", 30);
System.out.println("Primero: " + par.getPrimero()); // Imprime: Primero: Edad
System.out.println("Segundo: " + par.getSegundo()); // Imprime: Segundo: 30
```

```
public class Par<T, U> {
    private T primero;
    private U segundo;

    public Par(T primero, U segundo) {
        this.primero = primero;
        this.segundo = segundo;
    }

    public T getPrimero() {
        return primero;
    }

    public void setPrimero(T primero) {
        this.primero = primero;
    }

    public U getSegundo() {
        return segundo;
    }

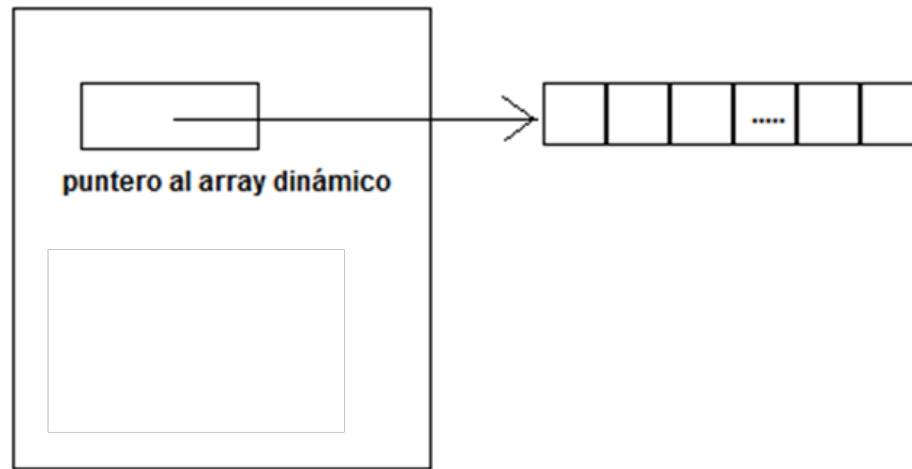
    public void setSegundo(U segundo) {
        this.segundo = segundo;
    }
}
```

# Estructuras de Datos lineales

- ▶ Vector
- ▶ Vector con template y redimensión
- ▶ Lista simplemente enlazada
- ▶ Lista con Template
- ▶ Lista con Cursor
- ▶ Lista doblemente enlazada
- ▶ Lista circular
- ▶ Lista circular doblemente enlazada
- ▶ Pila
- ▶ Cola

## Vector

- ▶ Almacenar los datos en un array dinámico.
- ▶ La instancia de Array es un objeto que tiene estos atributos:
- ▶ Un puntero al array dinámico
- ▶ Un entero indicando el tamaño del array





## Implementación 1:

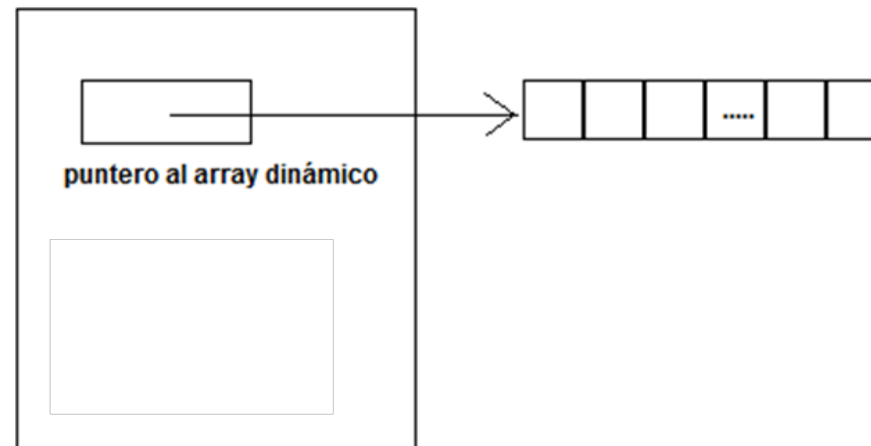
Las primitivas deben hacer lo siguiente:

Constructor: solicita y obtiene espacio para el array dinámico. Inicializa el atributo tamaño.

Destructor: destruye el array dinámico

setValor: recibe un valor y una posición y asigna a la posición dada el valor indicado

getValor: recibe una posición y retorna el valor de esa posición



```

3 public class Vector<T> {
4 //ATRIBUTOS DE CLASE -----
5 //ATRIBUTOS -----
6
7     private T[] datos = null;
8     private T datoInicial;
9
10 //CONSTRUCTORES -----
11
12 /**
13  * pre:
14  * @param longitud: entero mayor a 0, determina la cantidad de elementos del vector
15  * @param datoInicial: valor inicial para las posiciones del vector
16  * @throws Exception: da error si la longitud es invalida
17  * post: inicializa el vector de longitud de largo y todos los valores inicializados
18  */
19 Vector(int longitud, T datoInicial) throws Exception {
20     if (longitud < 1) {
21         throw new Exception("La longitud debe ser mayor o igual a 1");
22     }
23     this.datos = crearVector(longitud);
24     this.datoInicial = datoInicial;
25     for(int i = 0; i < this.getLongitud(); i++){
26         this.datos[i] = datoInicial;
27     }
28 }
29
30 //METODOS DE CLASE -----
31 //METODOS GENERALES -----
32 //METODOS DE COMPORTAMIENTO -----
33

```

```

34  /**
35   * pre:
36   * @param posicion: valor entre 1 y el largo del vector
37   * @param dato: -
38   * @throws Exception: da error si la posicion no esta en rango
39   * post: guarda la el dato en la posicion dada
40   */
41  void agregar(int posicion, T dato) throws Exception {
42      validarPosicion(posicion);
43      this.datos[posicion - 1] = dato;
44  }
45
46  /**
47   * pre: -
48   * @param posicion: valor entre 1 y el largo del vector
49   * @return devuelve el valor en esa posicion
50   * @throws Exception: da error si la posicion no esta en rango
51   */
52  T obtener(int posicion) throws Exception {
53      validarPosicion(posicion);
54      return this.datos[posicion - 1];
55  }
56

```

```

57  * pre: -
58  * @param posicion: valor entre 1 y el largo del vector
59  * @throws Exception: da error si la posicion no esta en rango
60  * post: remueve el valor en la posicion y deja el valor inicial
61  */
62
63 void remover(int posicion) throws Exception {
64     if ((posicion < 1) ||
65         (posicion > this.getLongitud())) {
66         throw new Exception("La " + posicion + " no esta en el rango 1 y " + this.getLongitud() + " inclusive");
67     }
68     this.datos[posicion - 1] = this.datoInicial;
69 }
70
71 /**
72  * pre:
73  * @param dato: valor a guardar
74  * @return devuelve la posicion en que se guardo
75  * @throws Exception
76  * post: guarda el dato en la siguiente posicion vacia
77  */
78 int agregar(T dato) throws Exception {
79     //validar dato;
80     for(int i = 0; i < this.getLongitud(); i++) {
81         if (this.datos[i] == this.datoInicial) {
82             this.datos[i] = dato;
83             return i + 1;
84         }
85     }
86     T[] temp = crearVector(this.getLongitud() * 2);
87     for(int i = 0; i < this.getLongitud(); i++) {
88         temp[i] = this.datos[i];
89     }
90     this.datos = temp;
91     this.datos[this.getLongitud() + 1] = dato;
92     return (this.getLongitud() / 2) + 1;
93 }

```

```

70
71- /**
72  * pre:
73  * @param dato: valor a guardar
74  * @return devuelve la posicion en que se guardo
75  * @throws Exception
76  * post: guarda el dato en la siguiente posicion vacia
77  */
78- int agregar(T dato) throws Exception {
79  //validar dato;
80  for(int i = 0; i < this.getLongitud(); i++) {
81      if (this.datos[i] == this.datoInicial) {
82          this.datos[i] = dato;
83          return i + 1;
84      }
85  }
86  throw new Exception("No hay lugar en el vector");|
87  }
88

```

```

94  /**
95  * pre: -
96  * @param posicion: valor entre 1 y el largo del vector
97  * @throws Exception: da error si la posicion no esta en rango
98  * post: valida la posicion que este en rango
99  */
100
101 private void validarPosicion(int posicion) throws Exception {
102     if ((posicion < 1) ||
103         (posicion > this.getLongitud())) {
104         throw new Exception("La " + posicion + " no esta en el rango 1 y " + this.getLongitud() + " inclusive");
105     }
106 }
107
108 /**
109 * pre:
110 * @param longitud: -
111 * @return devuelve un vector del tipo y longitud deseado
112 * @throws Exception
113 */
114 @SuppressWarnings("unchecked")
115 private T[] crearVector(int longitud) throws Exception {
116     if (longitud <= 0) {
117         throw new Exception("La longitud debe ser mayor o igual a 1");
118     }
119     return (T[]) new Object[longitud];
120 }
121
122 //GETTERS SIMPLES -----
123
124 public int getLongitud() {
125     return this.datos.length;
126 }
127

```

## Vector con redimensión

Los criterios de redimensión en estructuras en arreglo se refieren a las condiciones o reglas que determinan cuándo y cómo se ajusta el tamaño de un arreglo dinámico. Aquí hay algunos criterios comunes:

1. **Capacidad máxima:** Se establece un límite superior para el tamaño del arreglo. Cuando el arreglo alcanza esta capacidad máxima, se redimensiona para aumentar su tamaño según sea necesario.
2. **Factor de carga:** Se define un factor de carga que indica cuán lleno está el arreglo en relación con su capacidad total. Cuando el factor de carga supera un umbral predefinido (por ejemplo, 0.7), se redimensiona el arreglo para evitar un exceso de congestión y para mantener un rendimiento aceptable.
3. **Incremento de tamaño:** Cuando se necesita redimensionar el arreglo, se incrementa su tamaño en una cantidad fija o en un porcentaje específico. Por ejemplo, podría aumentarse en una cantidad fija de elementos o en un 50% de su tamaño actual.
4. **Reducción de tamaño:** En algunos casos, especialmente cuando la memoria es un recurso crítico, se puede implementar la reducción del tamaño del arreglo cuando su capacidad se vuelve significativamente mayor que la cantidad de elementos almacenados. Esto se hace para liberar la memoria no utilizada.
5. **Frecuencia de redimensionamiento:** Se establece un intervalo de tiempo o una frecuencia de operaciones en las cuales se verifica si el arreglo necesita redimensionarse. Esto puede ayudar a evitar redimensionamientos innecesarios y costosos.

## Redimensión por tamaño máximo

```
/**
 * pre:
 * @param dato: valor a guardar
 * @return devuelve la posicion en que se guardo
 * @throws Exception
 * post: guarda el dato en la siguiente posicion vacia
 */
int agregar(T dato) throws Exception {
    //validar dato;
    for(int i = 0; i < this.getLongitud(); i++) {
        if (this.datos[i] == this.datoInicial) {
            this.datos[i] = dato;
            return i + 1;
        }
    }
    T[] temp = crearVector(this.getLongitud() * 2);
    for(int i = 0; i < this.getLongitud(); i++) {
        temp[i] = this.datos[i];
    }
    this.datos = temp;
    this.datos[this.getLongitud() + 1] = dato;
    return (this.getLongitud() / 2) + 1;
}
```



# Listas

## Definición:

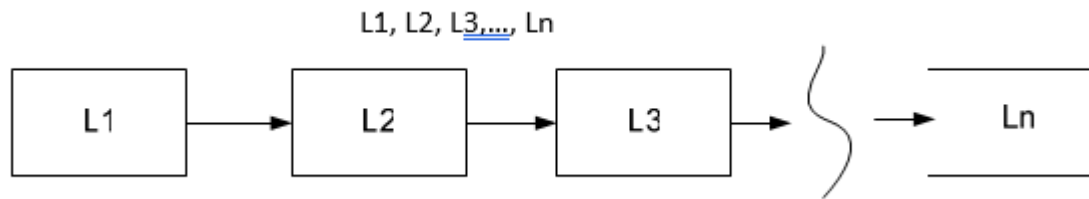
Una lista es una estructura de datos lineal flexible, ya que puede crecer (a medida que se insertan nuevos elementos) o acortarse (a medida que se borran elementos) según las necesidades que se presenten.

En principio, los elementos deben ser del mismo tipo (homogéneos) pero, cuando se estudie herencia y polimorfismo, podremos trabajar con elementos distintos, siempre y cuando hereden de algún antecesor común.

Los elementos pueden insertarse en cualquier posición de la lista, ya sea al principio, al final o en cualquier posición intermedia. Lo mismo sucede con el borrado.

## Propiedades - Representación


Matemáticamente, una lista es una secuencia ordenada de  $n$  elementos, con  $n \geq 0$  (si  $n = 0$  la lista se encuentra vacía) que podemos representar de la siguiente forma:



En la representación gráfica, las flechas no tienen por qué corresponder a punteros, aunque podrían serlo, se deben tomar como la indicación de secuencia ordenada.

Interfaz esperada:

- 1) Poder construirla / destruirla adecuadamente
- 2) Preguntar si esta vacia
- 3) Preguntar la cantidad de elementos
- 4) Agregar elementos
- 5) Cambiar elementos
- 6) Obtener elementos
- 7) Cambiar elementos
- 8) Quitar elementos
- 9) Recorrerla

Lista	
 CP	DIFERENTES IMPLEMENTACIONES
<hr/>	
Constructor	
Destructor	
estaVacia	
contarElementos	
agregar	
obtener	
asignar	
remove	
recorrido	

## Operaciones básicas

Como en todo tipo de dato abstracto (TDA) debemos definir un conjunto de operaciones que trabajen con el tipo lista. Estas operaciones no siempre serán adecuadas para cualquier aplicación. Por ejemplo, si queremos insertar un cierto elemento  $x$  debemos decidir si la lista permite o no tener elementos duplicados.

Las operaciones básicas que deberá manejar una lista son las siguientes:

- Insertar un elemento en la lista (transformación).
- Borrar un elemento de la lista (transformación).
- Obtener un elemento de la lista (observación).

Las dos primeras operaciones transformarán la lista, ya sea agregando un elemento o quitándolo. La última operación sólo será de inspección u observación, consultando por un determinado elemento pero sin modificar la lista.

Obviamente, además se necesitará, como en todo TDA, una operación de creación y otra de destrucción.

A continuación, analicemos con mayor profundidad cada una de las operaciones mencionadas.

## Insertar

La operación insertar puede tener alguna de estas formas:

- i. insertar (x)
- ii. insertar (x, p)

En el primer caso (i), la lista podría mantenerse ordenada por algún campo clave, con lo cual, se compararía  $x.clave$  con los campos clave de los elementos de la lista, insertándose  $x$  en el lugar correspondiente.

Pero, también, podría ser una lista en la que no interesa guardar ningún orden en especial, por lo que la inserción podría realizarse en cualquier lugar, en particular podría ser siempre al principio o al final (sólo por una comodidad de la implementación).

En el segundo caso (ii),  $p$  representa la posición en la que debe insertarse el elemento  $x$ . Por ejemplo, en una lista

$L_1, L_2, \dots, L_n$

insertar ( $x, 1$ ) produciría el siguiente resultado:

$x, L_1, L_2, \dots, L_n$

en cambio insertar ( $x, n+1$ ) agregaría a  $x$  al final de la lista.

En esta operación se debe decidir qué hacer si  $p$  supera la cantidad de elementos de la lista en más de uno

$p > n + 1$

Las decisiones podrían ser varias, desde no realizar nada (no insertar ningún elemento), insertarlo al final o estipular una precondition del método en la que esta situación no pueda darse nunca. De esta última forma, se transfiere la responsabilidad al usuario del TDA, quien debería cuidar que nunca se dé esa circunstancia.

## Borrar

En el borrado de un elemento pasa una situación similar a la que se analizó en la inserción.

Las operaciones podrían ser

- i. eliminar (  $x$  )
- ii. eliminar (  $p$  )

La primera (i) borra el elemento  $x$  de la lista. Si no lo encontrara podría, simplemente, no hacer nada. En la segunda (ii), borra el elemento de la lista que se encuentra en la posición

$p$ . Las decisiones en cuanto a si  $p$  es mayor que  $n$  (la cantidad de elementos de la lista) son las mismas que en el apartado anterior, es decir, no hacer nada o borrar el último elemento.

## Obtener

Este método debe recuperar un elemento determinado de la lista. Las opciones son las siguientes:

- i. getElemento ( x.clave)
- ii. getElemento (p)

En la opción uno (i) se tiene una parte del elemento, sólo la clave, y se desea recuperarlo en su totalidad. En cambio, en la opción dos (ii) se desea recuperar el elemento que está en la posición p.

En ambas opciones, el resultado debe devolverse. Es decir, el método debería devolver el elemento buscado. Sin embargo, esto no sería consistente en el caso de no encontrarlo.

¿Qué devolvería el método si la clave no se encuentra? Por otro lado, devolver un objeto no siempre es lo ideal, ya que se debería hacer una copia del elemento que está en la lista, lo que sería costoso y difícil de implementar en muchos casos. Por estos motivos se prefiere devolver un puntero o referencia al objeto de la lista. De esta forma se ahorra tiempo, espacio y, si el elemento no se encontrara en la lista, se puede devolver un valor nulo (NULL).

Por supuesto puede haber más operaciones que las mencionadas, pero éstas son las básicas, a partir de las cuales se pueden obtener otras. Por ejemplo, se podría desear tener un borrado completo de la lista, pero esta operación se podría realizar llamando al borrado del primer elemento hasta que la lista quedara vacía.

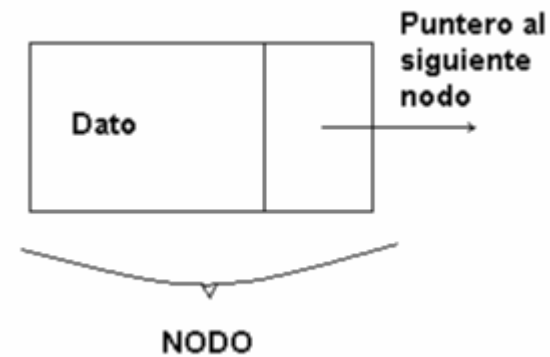
## Implementacion de Lista con estructuras dinámicas

### Lista simplemente enlazada

Antes de hablar de listas dinámicas debemos hablar de nodos. Las listas enlazarán nodos. Un nodo es una estructura que tendrá los siguientes datos:

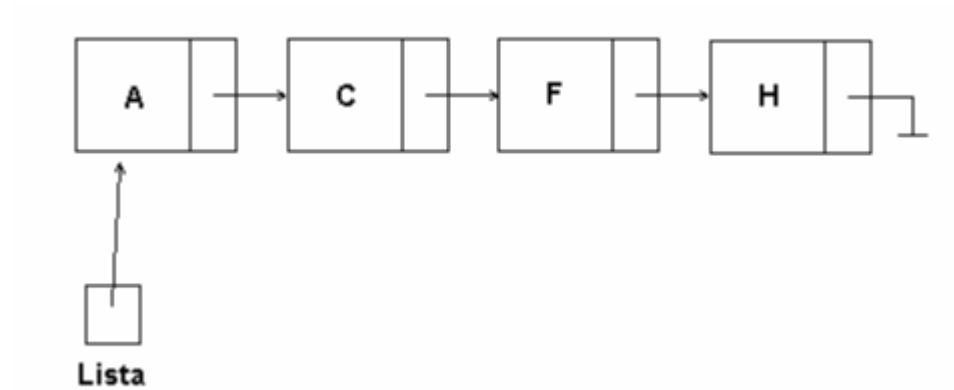
- El propio elemento que deseamos almacenar.
- Uno o más enlaces a otros nodos (punteros, con las direcciones de los nodos enlazados).

En el caso más simple, el nodo tendrá el elemento y un puntero al siguiente nodo de la lista:

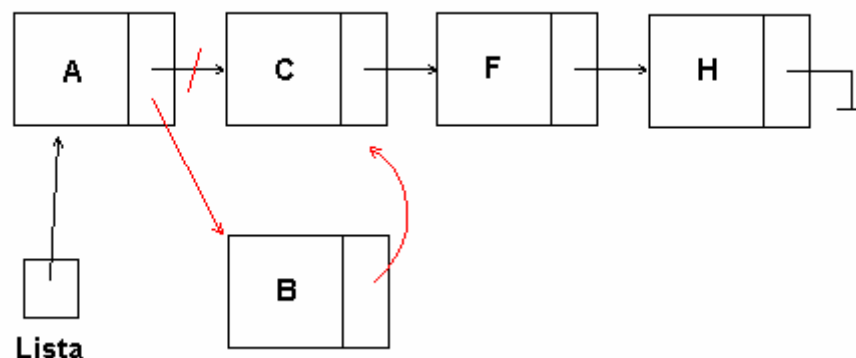




Para implementarlo, necesitamos un puntero al primer nodo de la lista, el resto se irán enlazando mediante sus propios punteros. El último puntero, apuntará a nulo y se representa como un "cable a tierra".



La inserción es una operación muy elemental (no de codificar, sino en costos de tiempos, una vez ubicado el lugar donde se insertará). En este caso, si queremos insertar el elemento 'B' sólo tendremos que ajustar dos punteros:



Se reasigna el puntero que tiene el nodo donde está 'A', apuntando al nuevo nodo, que contiene 'B'. El puntero del nodo de 'B' apuntará al mismo lugar donde apuntaba 'A', que es 'C'.

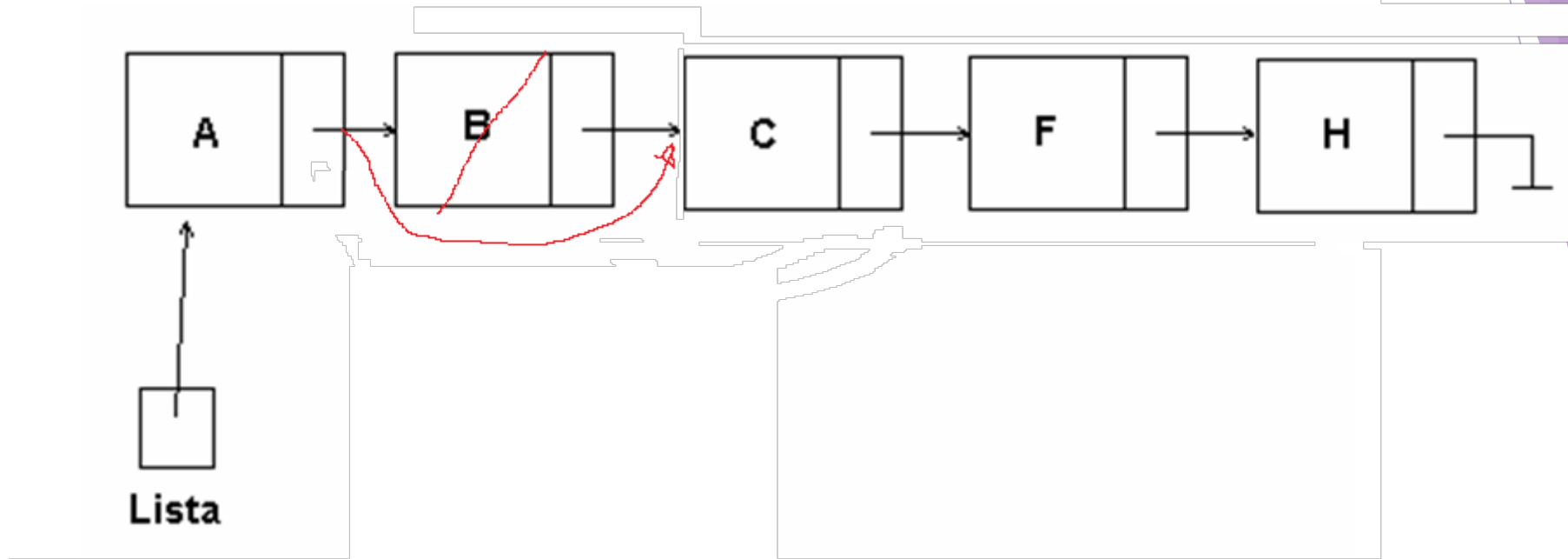
Estos nodos se irán creando en tiempo de ejecución, en forma dinámica, a medida que se vayan necesitando.

Si bien esta implementación necesita más memoria que la estática, ya que, además del dato, debemos almacenar un puntero, no hay desperdicio, debido a que sólo se crearán los nodos estrictamente necesarios.

A modo de ejemplo, en el apéndice A se muestra una implementación muy básica (apenas una modificación de la anterior) de esta estructura. Cabe destacar que, a estas alturas, se debe agregar una nueva clase que es Nodo.

Como se utiliza memoria dinámica cobra especial importancia el destructor, y hay que tener cuidado de liberar la memoria cada vez que se borra un nodo.

La eliminacion de la posicion o de la clave en la lista, varia si es el primer nodo o si es un nodo intermedio



Ubica el nodo anterior, asigna como siguiente nodo el siguiente del nodo a eliminar y luego hace el delete del nodo a eliminar.

## Lista con template

Las listas, al igual que otras estructuras, siempre operan de la misma manera sin importarle el tipo de dato que estén albergando. Por ejemplo, agregar un elemento al final o borrar el tercer nodo deberá tener el mismo algoritmo ya sea que el elemento fuera un char, un float o una estructura. Sin embargo, cuando uno define los métodos debe indicar de qué tipo son los parámetros y de qué tipo van a ser algunas devoluciones.

Sería muy tonto repetir varias veces el mismo código sólo para cambiar una palabra:

char por float o por registro\_empleado, por ejemplo.

En el apartado anterior vimos que esto se solucionaba utilizando un puntero genérico (void\*), sin embargo, un problema se resolvía pero se introducía uno nuevo: la falta de control en los tipos. Pero este problema no es el único, ya que uno podría decidir prescindir del control de tipos a cambio de un mayor cuidado en la codificación, por ejemplo. Otro problema muy importante es que un puntero a void no nos permite utilizar operadores, ya que el compilador no sabe a qué tipo de dato se lo estaría aplicando. Por ejemplo, el operador "+" actúa en forma muy distinta si los argumentos son números enteros (los suma) que si fueran strings (los concatena).

Lo curioso es que, en el ejemplo de uso anterior, en la función main uno tenía en claro qué tipo de dato estaba colocando en la lista (primero se colocaron direcciones de enteros y, en otra, strings) pero, en el momento de ingresar a la lista, pierden esa identidad, ya que se toman como direcciones a void. A partir de ahí uno pierde el uso de operadores, como el + o el - y, también pierde el uso de otros métodos.

Otra solución, que verán más adelante, es la utilización de la herencia y el polimorfismo.

El último de los enfoques, teniendo en la mira el objetivo de la programación genérica, son las plantillas (templates).

Los templates, en lugar de reutilizar el código objeto, como se estudiará en el polimorfismo, reutiliza el código fuente. ¿De qué manera? Con parámetros de tipo no especificado. Veamos cómo se hace esto modificando nuestra implementación de Lista\_estatica.

## Lista de enteros

Como primera aproximación vamos a hacer una lista especifica de enteros:

```
-
3 public class Lista {
4     //ATRIBUTOS DE CLASE -----
5     //ATRIBUTOS -----
6
7     private Nodo primero = null;
8     private int tamanio = 0;
9
10    //CONSTRUCTORES -----
11
12    /**
13     * pre: -
14     * post: crea una lista vacia
15     */
16    public Lista() {
17        this.primerο = null;
18        this.tamanio = 0;
19    }
20
21    //METODOS DE CLASE -----
22    //METODOS GENERALES -----
23    //METODOS DE COMPORTAMIENTO -----
24
```

```
//METODOS DE CLASE -----  
//METODOS GENERALES -----  
//METODOS DE COMPORTAMIENTO -----
```

```
/**  
 * pre:  
 * @param elemento: -  
 * @throws Exception  
 * post: agrega un elemento al final de la lista  
 */  
public void agregar(int elemento) throws Exception {  
    this.agregar(elemento, this.getTamanio() + 1);  
}  
  
/**  
 * pre: -  
 * @return devuelve verdadero si la lista esta vacia  
 */  
public boolean estaVacia() {  
    return (this.tamanio == 0);  
}
```

```
/**  
 * pre: -  
 * @param elemento: -  
 * @param posicion: debe estar entre 1 y el tamaño  
 * @throws Exception: da error si la posicion no esta en rango  
 * post: agrega un elemento en la posicion indicada  
 */  
public void agregar(int elemento, int posicion) throws Exception {  
    validarPosicion(posicion);  
    Nodo nuevo = new Nodo(elemento);  
    if (posicion == 1) {  
        nuevo.setSiguiete( this.primer);  
        this.primer = nuevo;  
    } else {  
        Nodo anterior = this.obtenerNodo(posicion -1);  
        nuevo.setSiguiete( anterior.getSiguiente());  
        anterior.setSiguiete( nuevo );  
    }  
    this.tamanio++;  
}  
  
/**  
 *  
 * @param posicion  
 * @return  
 */  
private Nodo obtenerNodo(int posicion) {  
    //validarPosicion(posicion);  
    Nodo actual = this.primer;  
    for(int i = 1; i < posicion; i++) {  
        actual = actual.getSiguiente();  
    }  
    return actual;  
}
```

```

64 /**
65  *
66  * @param posicion
67  * @return
68  */
69 private Nodo obtenerNodo(int posicion) {
70     //validarPosicion(posicion);
71     Nodo actual = this.primerO;
72     for(int i = 1; i < posicion; i++) {
73         actual = actual.getSiguiente();
74     }
75     return actual;
76 }
77
78 /**
79  * pre:
80  * @param posicion: en rango
81  * @throws Exception
82  */
83 private void validarPosicion(int posicion) throws Exception {
84     if ((posicion < 1) ||
85         (posicion > this.tamano + 1)) {
86         throw new Exception("La posicion debe estar " +
87             "entre 1 y tamaño + 1");
88     }
89 }
90

```

```

/**
 * pre:
 * @param posicion: en rango de 1 a tamaño
 * @throws Exception
 */
public void remover(int posicion) throws Exception {
    validarPosicion(posicion);
    Nodo removido;
    if (posicion == 1) {
        removido = this.primerO;
        this.primerO = removido.getSiguiente();
    } else {
        Nodo anterior = this.obtenerNodo(posicion - 1);
        removido = anterior.getSiguiente();
        anterior.setSiguiente(removido.getSiguiente());
    }
    this.tamano--;
}

//GETTERS SIMPLES -----

public int getTamano() {
    return this.tamano;
}

int getDato(int posicion) throws Exception {
    validarPosicion(posicion);
    return this.obtenerNodo(posicion).getDato();
}

//SETTERS SIMPLES -----

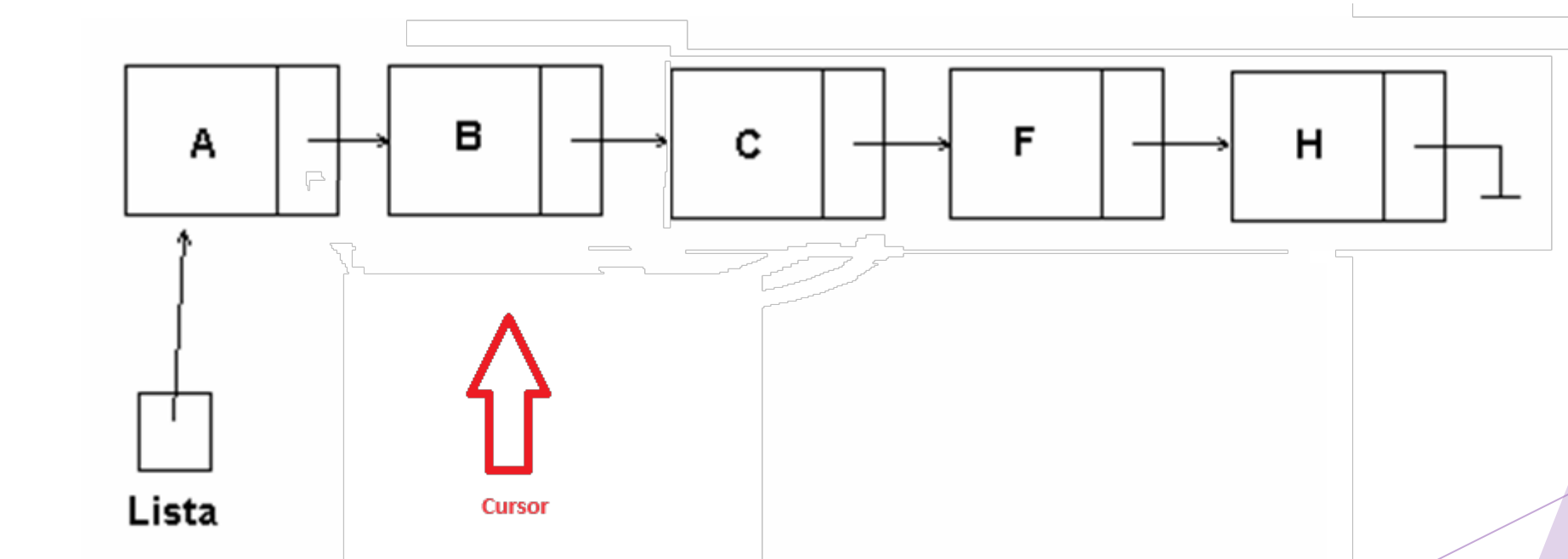
void setDato(int elemento, int posicion) throws Exception {
    validarPosicion(posicion);
    this.obtenerNodo(posicion).setDato(elemento);
}

```

## Lista con Cursor

La lista es estructura que se recorre con un while, pero para hacer el seguimiento se hace con un puntero al nodo actual, que se llama cursor.

Cuando se inicia el recorrido, este queda apuntando a NULL. Luego se avanza y se va obteniendo el dato del cursor hasta llegar al final.





```

3 public class Lista<T> {
4     //ATRIBUTOS DE CLASE -----
5     //ATRIBUTOS -----
6
7     private Nodo<T> primero = null;
8     private int tamano = 0;
9     private Nodo<T> cursor = null;
10
11     //CONSTRUCTORES -----
12
13     /**
14      * pre:
15      * pos: crea una lista vacia
16      */
17     public Lista() {
18         this.primero = null;
19         this.tamano = 0;
20         this.cursor = null;
21     }
22
23     //METODOS DE CLASE -----
24     //METODOS GENERALES -----
25     //METODOS DE COMPORTAMIENTO -----
26
27     /**
28      * pre:
29      * pos: indica si la lista tiene algún elemento.
30      */
31     public boolean estaVacia() {
32         return (this.tamano == 0);
33     }
34

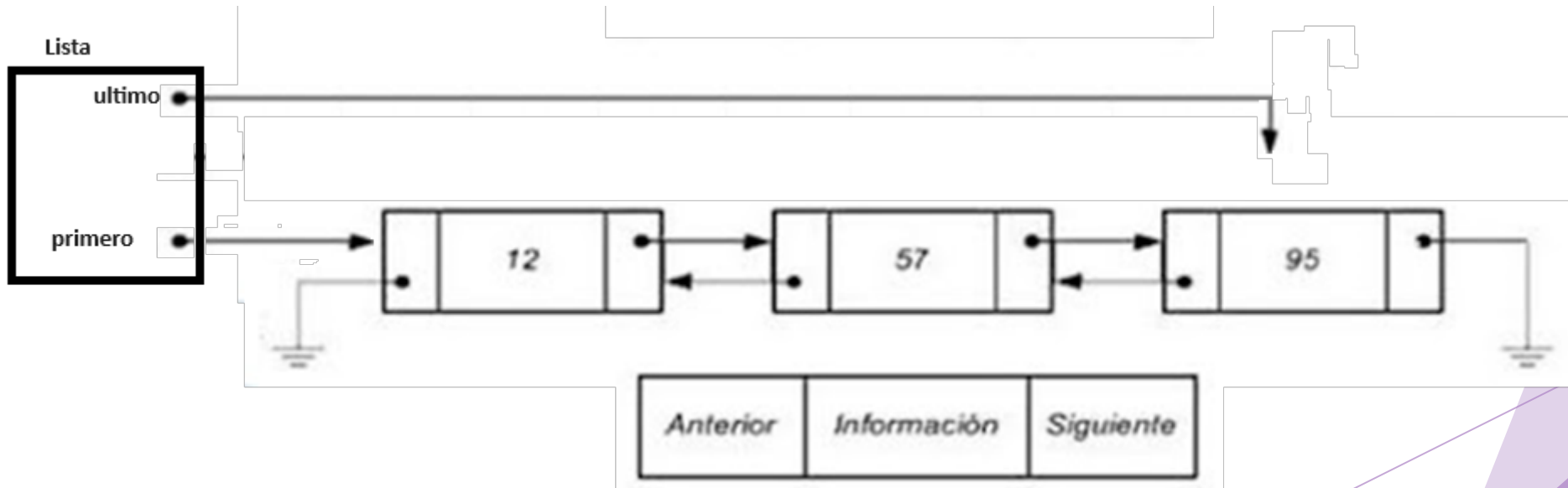
```

## Recorrido de la lista

```
16
17 public static void main(String[] args) {
18
19     Lista<Integer> lista = new Lista<Integer>();
20     lista.iniciarCursor();
21     while (lista.avanzarCursor()) {
22         System.out.println("El numero es: " + lista.obtenerCursor());
23     }
24 }
25
26 }
27
```

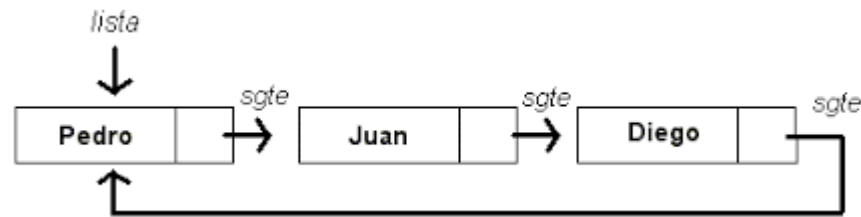
## Lista doblemente enlazada

La lista doblemente enlazada es una estructura de datos que consiste en un conjunto de nodos enlazados secuencialmente. Cada nodo contiene tres campos, dos para los llamados enlaces, que son referencias al nodo siguiente y al anterior en la secuencia de nodos, y otro más para el almacenamiento de la información. El enlace al nodo anterior del primer nodo y el enlace al nodo siguiente del último nodo, apuntan a un tipo de nodo que marca el final de la lista, normalmente un puntero NULL, para facilitar el recorrido de la lista.

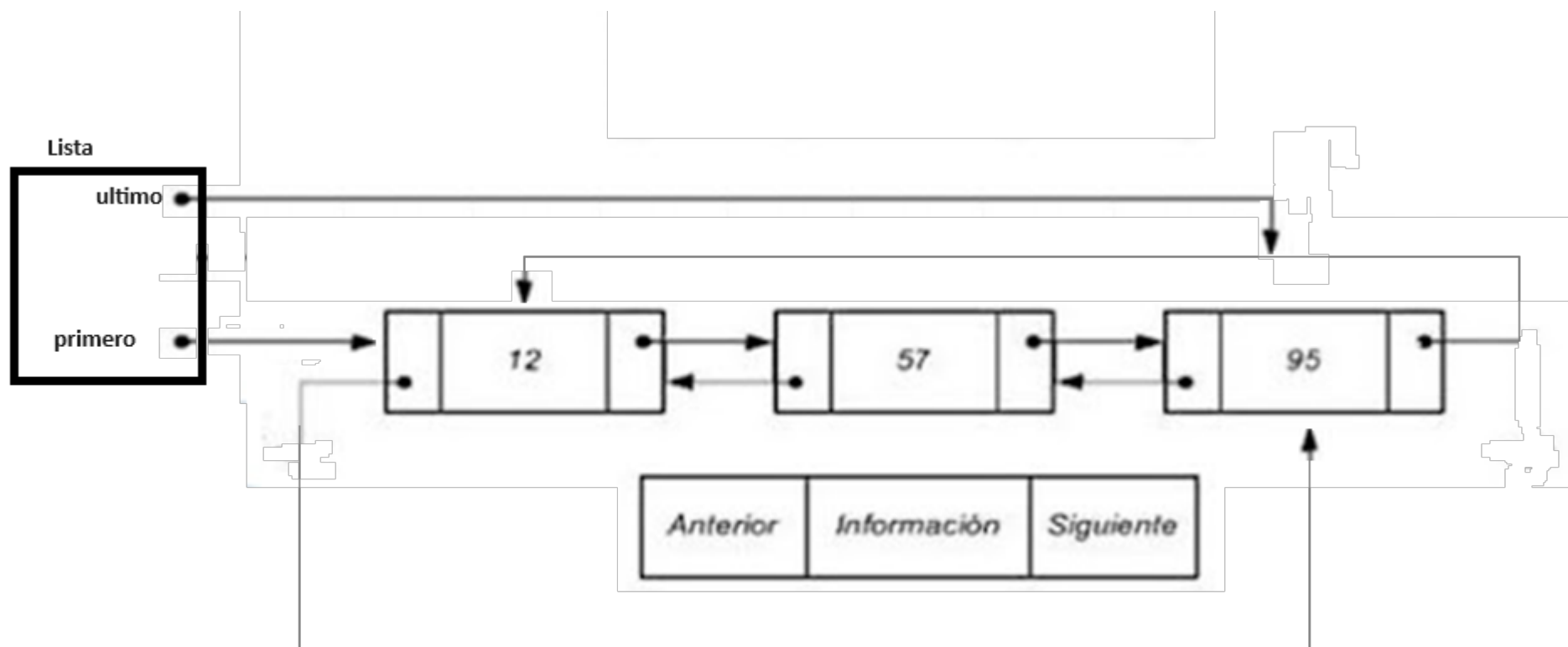


## Lista circular

Una lista circular es una lista lineal en la que el último nodo apunta al primero. Las listas circulares evitan excepciones en las operaciones que se realicen sobre ellas. No existen casos especiales, cada nodo siempre tiene uno anterior y uno siguiente.



## Lista circular doblemente enlazada



## Pila

Una pila es un contenedor de objetos que se insertan y se eliminan siguiendo el principio 'Último en entrar, primero en salir' (L.I.F.O.= 'Last In, First Out'). a característica más importante de las pilas es su forma de acceso.

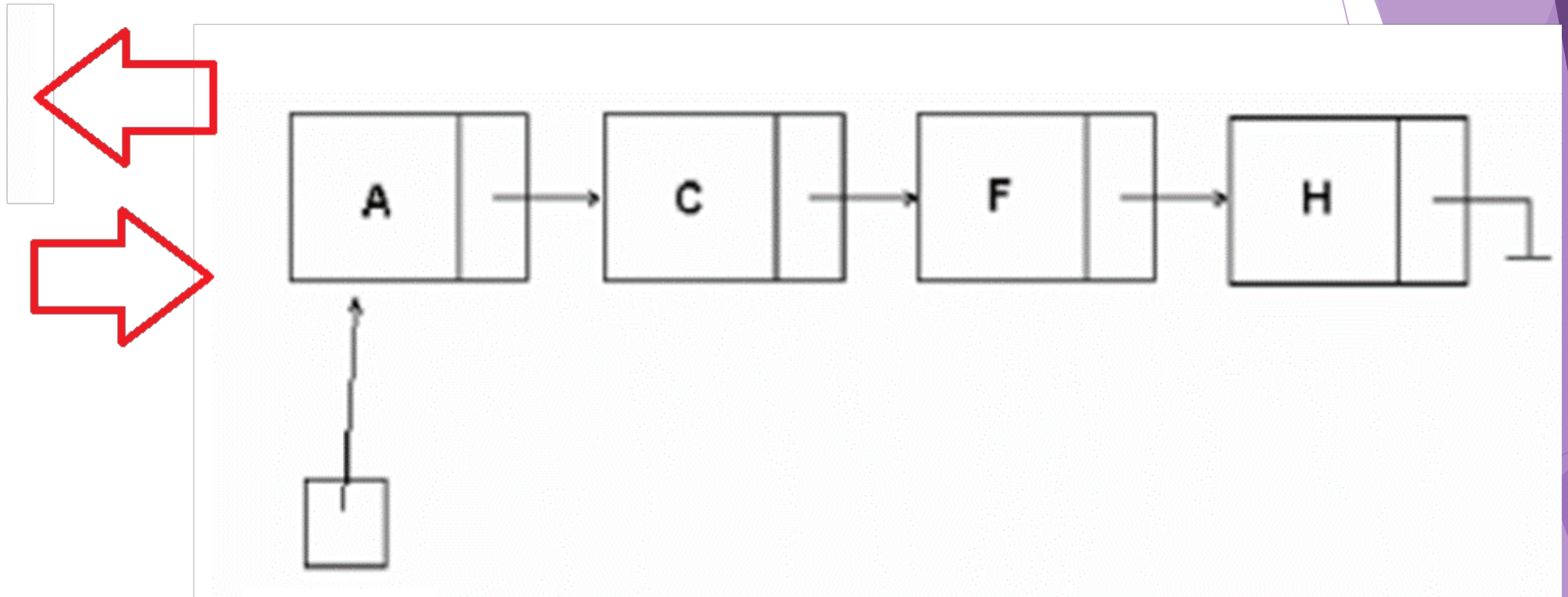
En ese sentido puede decirse que una pila es una clase especial de lista en la cual todas las inserciones (alta ,apilar o push) y borrados (baja, desapilar o pop) tienen lugar en un extremo denominado cabeza o tope.

El Tope de la pila corresponde al elemento que entró en último lugar, es decir que saldrá en la próxima baja.

En lenguajes de alto nivel, es muy importante para la la eliminación de la recursividad, análisis de expresiones, etc.

En lenguajes de bajo nivel es indispensable y actúa constantemente en todos los lenguajes compilados y en todo el sistema operativo.

Se puede considerar una pila como una estructura ideal e infinita, o bien como una estructura finita



```

3 public class Pila<T> {
4     //ATRIBUTOS DE CLASE -----
5     //ATRIBUTOS -----
6
7     private Nodo<T> tope = null;
8     private int tamano = 0;
9
10    //CONSTRUCTORES -----
11
12    /**
13     * pre:
14     * post: inicializa la pila vacia para su uso
15     */
16    public Pila() {
17        this.tope = null;
18        this.tamano = 0;
19    }
20
21    //METODOS DE CLASE -----
22    //METODOS GENERALES -----
23    //METODOS DE COMPORTAMIENTO -----
24
25    /**
26     * post: indica si la cola tiene algún elemento.
27     */
28    public boolean estaVacia() {
29        return (this.tamano == 0);
30    }
31
32    /**
33     * pre: el elemento no es vacio
34     * post: agrega el elemento a la pila
35     */
36    public void apilar(T elemento) {
37        Nodo<T>nuevo = new Nodo<T>(elemento);
38        nuevo.setSiguiente(this.tope);
39        this.tope = nuevo;
40    }
41
42    /**
43     * pre: el elemento no es vacio
44     * post: agrega el elemento a la pila
45     */
46    public void apilar(Lista<T> lista) {
47        //validar
48        lista.iniciarCursor();
49        while (!lista.avanzarCursor()) {
50            this.apilar(lista.obtenerCursor());
51        }
52    }
53
54    /**
55     * pre :
56     * post: devuelve el elemento en el tope de la pila y achica la pila en 1.
57     */
58    public T desapilar() {
59        T elemento = null;
60        if (!this.estaVacia()) {
61            elemento = this.tope.getDato();
62            Nodo<T> aBorrar = this.tope;
63            this.tope = this.tope.getSiguiente();
64        }
65        return elemento;
66    }

```



```

67
68= /**
69  * pre: -
70  * post: devuelve el elemento en el tope de la pila (solo lectura)
71  */
72= public T obtener() {
73     T elemento = null;
74     if (!this.estaVacia()) {
75         elemento = this.tope.getDato();
76     }
77     return elemento;
78 }
79
80= /**
81  * post: devuelve la cantidad de elementos que tiene la cola.
82  */
83= public int contarElementos() {
84     return this.tamano;
85 }
86
87 //GETTERS SIMPLES -----
88 //SETTERS SIMPLES -----
89
90 }

```

## Cola

Un TDA cola es un contenedor de objetos que se insertan y se eliminan siguiendo el principio 'Primero en entrar, primero en salir' (F.I.F.O.= 'First In, First Out').

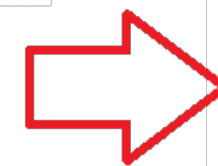
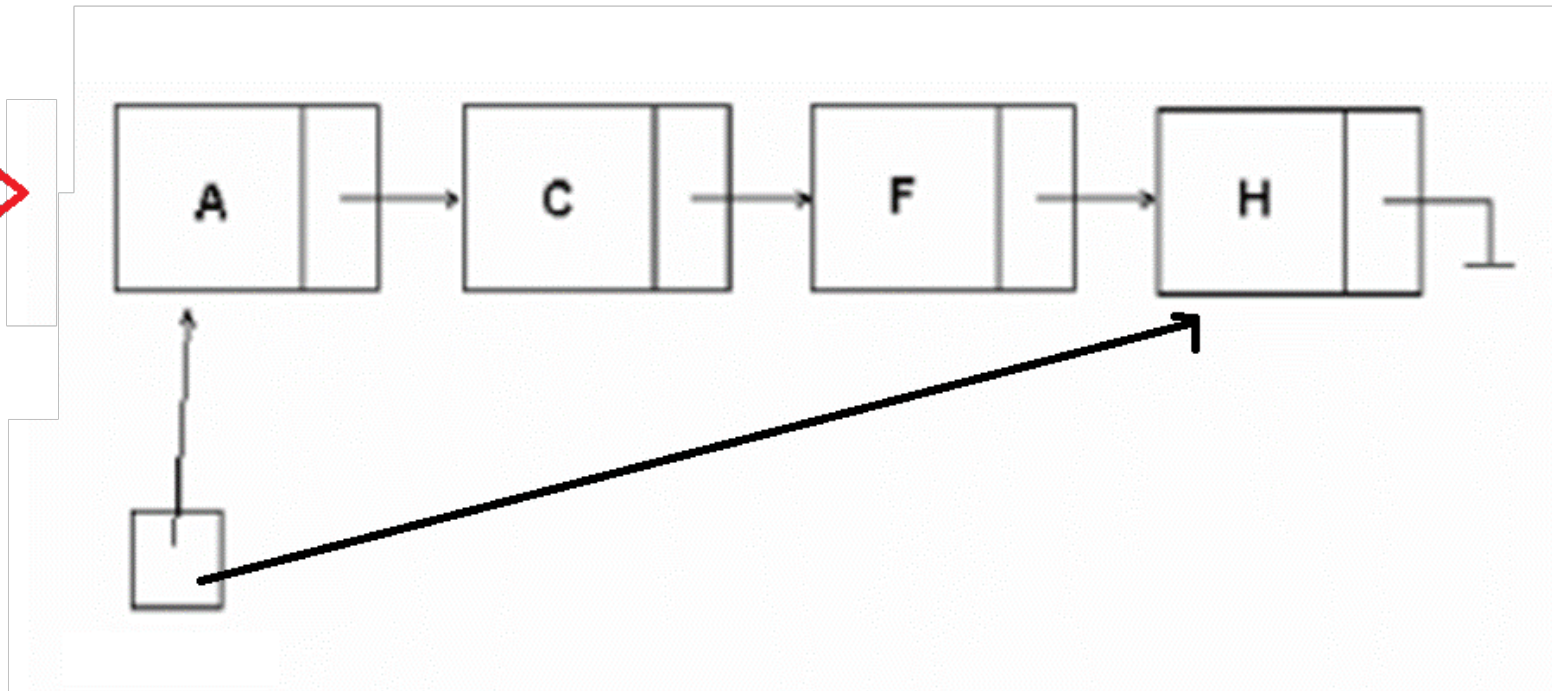
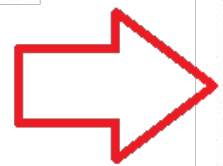
En ese sentido puede decirse que una cola es una lista con restricciones en el alta (encolar o enqueue) y baja (desencolar, dequeue).

El Frente (FRONT) de la cola corresponde al elemento que está en primer lugar, es decir que es el que estuvo más tiempo en espera.

El Fondo (END) de la cola corresponde al último elemento ingresado a la misma.

Se puede considerar una cola como una estructura ideal e infinita, o bien como una estructura finita.

El TDA cola se usa como parte de la solución de muchos problemas algorítmicos, y en situaciones tipo 'productor-consumidor', cuando una parte de un sistema produce algún elemento otra 'consume' más lentamente de lo que es producida, por lo cual los elementos esperan en una cola hasta ser consumidos.



```

3 public class Cola<T> {
4     //ATRIBUTOS DE CLASE -----
5     //ATRIBUTOS -----
6
7     private Nodo<T> frente = null;
8
9     private Nodo<T> ultimo = null;
10
11     private int tamaño = 0;
12
13     //CONSTRUCTORES -----
14
15     /**
16      * pre:
17      * post: inicializa la cola vacia para su uso
18      */
19     Cola() {
20         this.frente = null;
21         this.ultimo = null;
22         this.tamaño = 0;
23     }
24
25     //METODOS DE CLASE -----
26     //METODOS GENERALES -----
27     //METODOS DE COMPORTAMIENTO -----
28
29     /**
30      * post: indica si la cola tiene algún elemento.
31      */
32     public boolean estaVacia() {
33         return (this.tamaño == 0);
34     }
35

```

```

37  * pre: el elemento no es vacio
38  * post: agrega el elemento a la cola
39  */
40  public void acolar(T elemento) {
41      Nodo<T> nuevo = new Nodo<T>(elemento);
42      if (!this.estaVacia()) {
43          nuevo.setSiguiente(this.ultimo);
44          this.ultimo = nuevo;
45      } else {
46          this.frente = nuevo;
47          this.ultimo = nuevo;
48      }
49  }
50
51  /*
52  * pre: el elemento no es vacio
53  * post: agrega el elemento a la cola
54  */
55  void acolar(Lista<T> lista) {
56      //validar
57      lista.iniciarCursor();
58      while (!lista.avanzarCursor()) {
59          this.acolar(lista.obtenerCursor());
60      }
61  }
62

```

```

64  * pre :
65  * post: devuelve el elemento en el frente de la cola quitandolo.
66  */
67  public T desacolar() {
68      T elemento = null;
69      if (!this.estaVacia()) {
70          elemento = this.frente.getDato();
71          this.frente = this.frente.getSiguiente();
72      }
73      return elemento;
74  }
75
76  /*
77  * post: devuelve la cantidad de elementos que tiene la cola.
78  */
79  public int contarElementos() {
80
81      return this.tamanio;
82  }
83
84  //GETTERS SIMPLES -----
85
86  /*
87  * pre :
88  * post: devuelve el elemento en el frente de la cola. Solo lectura
89  */
90  public T obtener() {
91      T elemento = null;
92      if (!this.estaVacia()) {
93          elemento = this.frente.getDato();
94      }
95      return elemento;
96  }
97
98  //SETTERS SIMPLES -----
99  }

```

# Ejercicio 1

3. Implementar el método 'buscarSolucionesEquivalente' de la clase 'IngenieroQuimico' a partir de las siguientes especificaciones:

<pre>class IngenieroQuimico {      public:      /* post: busca en 'solucionesDisponibles' las Soluciones que tenga los mismos Compuestos que 'solucionRequerida',      *      con cantidades iguales o superiores pero menores (o igual) al doble.      */     Lista&lt;Solucion*&gt;* buscarSolucionesEquivalente(Solucion* solucionRequerida, Lista&lt;Solucion*&gt;*     solucionesDisponibles) };</pre>	
<pre>class Solucion {      public:      /* post: crea la solución con el código especificado, sin      *      compuestos asociados      */     Solucion(string codigo);      string obtenerCodigo();      /* post: devuelve los Compuestos requeridos para     preparar      *      la Solución      */     Lista&lt;Compuesto*&gt;* obtenerCompuestos() }; enum Unidad {     KILO; LITRO; };</pre>	<pre>class Compuesto {     public:      /* post: crea el Compuesto, identificado por 'nombre',      *      con cantidad 0 de la Unidad 'unidadDeMedida'      */     Compuesto(string nombre, Unidad unidadDeMedida);      /* post: devuelve el nombre que identifica el Compuesto      */     string obtenerNombre();      /* post: devuelve la Unidad en la que se mide la     cantidad.      */     Unidad obtenerUnidad();      float obtenerCantidad();      void cambiarCantidad(float cantidad); };</pre>

## Ejercicio 2

3. Implementar el método **seleccionarImagen** de la clase **Editor** a partir de las siguientes especificaciones:

```
class Editor {  
  
    public:  
  
    /* post: selecciona de 'imagenesDisponibles' aquella que tenga por lo menos tantos Comentarios como los indicados y  
     *      el promedio de calificaciones sea máximo. Ignora los Comentarios sin calificación.  
     */  
    Imagen* seleccionarImagen(Lista<Imagen*>* imagenesDisponibles, int cantidadDeComentarios);  
  
};  
  
class Imagen {  
  
    public:  
  
    /* post: inicializa la Imagen alojada en la URL indicada.  
     */  
    Imagen(string url);  
  
    /* post: devuelve la URL en la que está alojada.  
     */  
    string obtenerUrl();  
  
    /* post: devuelve los comentarios asociados.  
     */  
    Lista<Comentario*>* obtenerComentarios();  
  
    ~Imagen();  
};  
  
class Comentario {  
  
    public:  
  
    /* post: inicializa el Comentario con el contenido  
     *      y calificación 0.  
     */  
    Comentario(string contenido);  
  
    string obtenerContenido();  
  
    /* post: devuelve la calificación [1 a 10] asociada,  
     *      o 0 si el Comentario no tiene calificación.  
     */  
    int obtenerCalificacion();  
  
    /* pre : calificacion está comprendido entre 1 y 10  
     * post: cambia la calificación del Comentario.  
     */  
    void calificar(int calificacion);  
  
};
```

# Fin