

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;
    }

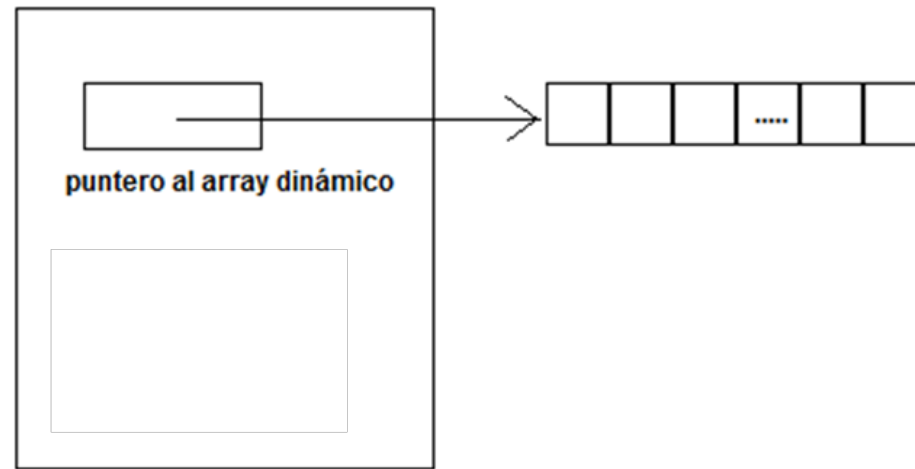
    public void setSegundo(U segundo) {
        this.segundo = segundo;
    }
}
```

Estructuras de Datos lineales

- ▶ Vector
- ▶ Vector con template y redimensión
- ▶ Lista simplemente enlazada
- ▶ Lista con Template
- ▶ Lista con Cursor (con template)
- ▶ Lista con iterador (con template)
- ▶ 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 que indique la cantidad de valores



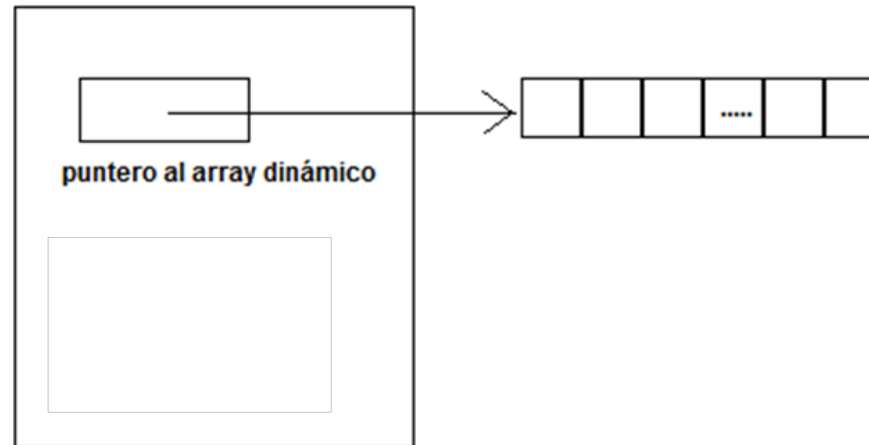
Implementación 1:

Las primitivas deben hacer lo siguiente:

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

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 VectorConEnteros {
4 //ATRIBUTOS DE CLASE -----
5 //ATRIBUTOS -----
6
7     private int [] datos = null;
8     private int cantidadDeDatos = 0;
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     public VectorConEnteros(int longitud) 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         for(int i = 0; i < this.getLongitud(); i++){
25             this.datos[i] = 0;
26         }
27     }
28 }

```

```

29 //METODOS DE CLASE -----
30 //METODOS GENERALES -----
31 //METODOS DE COMPORTAMIENTO -----
32
33 /**
34  * pre:
35  * @param posicion: valor entre 1 y el largo del vector
36  * @param dato: -
37  * @throws Exception: da error si la posicion no esta en rango
38  * post: guarda la el dato en la posicion dada
39  */
40 public void agregar(int posicion, int dato) throws Exception {
41     validarPosicion(posicion);
42     this.datos[posicion - 1] = dato; //falta redimensionar
43 }
44
45 /**
46  * pre: -
47  * @param posicion: valor entre 1 y el largo del vector
48  * @return devuelve el valor en esa posicion
49  * @throws Exception: da error si la posicion no esta en rango
50  */
51 public int obtener(int posicion) throws Exception {
52     validarPosicion(posicion);
53     return this.datos[posicion - 1];
54 }
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  public void remover(int posicion) throws Exception {
63      if ((posicion < 1) ||
64          (posicion > this.getLongitud())) {
65          throw new Exception("La " + posicion + " no esta en el rango 1 y " + this.getLongitud() + " inclusive");
66      }
67      this.datos[posicion - 1] = 0;
68  }
69
70  /**
71   * pre:
72   * @param dato: valor a guardar
73   * @return devuelve la posicion en que se guardo
74   * @throws Exception
75   * post: guarda el dato en la siguiente posicion vacia
76   */
77  public int agregar(int dato) throws Exception {
78      if (this.cantidadDeDatos < this.getLongitud()) {
79          this.datos[this.cantidadDeDatos] = dato;
80          this.cantidadDeDatos++;
81          return this.cantidadDeDatos;
82      }
83      throw new Exception("No hay mas lugar");
84  }
85

```

```

86-  /**
87-   * pre: -
88-   * @param posicion: valor entre 1 y el largo del vector
89-   * @throws Exception: da error si la posicion no esta en rango
90-   * post: valida la posicion que este en rango
91-   */
92- private void validarPosicion(int posicion) throws Exception {
93-     if ((posicion < 1) ||
94-         (posicion > this.getLongitud())) {
95-         throw new Exception("La " + posicion + " no esta en el rango 1 y " + this.getLongitud() + " inclusive");
96-     }
97- }
98-
99- /**
100-  * pre:
101-  * @param longitud: -
102-  * @return devuelve un vector del tipo y longitud deseado
103-  * @throws Exception
104-  */
105- private int[] crearVector(int longitud) throws Exception {
106-     if (longitud <= 0) {
107-         throw new Exception("La longitud debe ser mayor o igual a 1");
108-     }
109-     return new int[longitud];
110- }
111-
112- //GETTERS SIMPLES -----
113-
114- public int getLongitud() {
115-     return this.datos.length;
116- }
117-
118- //SETTERS SIMPLES -----
119-
120- }

```

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.

Vector con template y redimension

Combinando el TDA de Vector de enteros con la sintaxis de template, generamos un TDA Vector de colección dinámica en Java para almacenar cualquier tipo de datos.

A este TDA Vector le agregamos la funcionalidad de Redimensionamiento automático, que permite que el Vector crezca a medida que se van agregando elementos.

A este vector, se puede implementar con cualquiera de las opciones de redimensionamiento vistas, modificando el método agregar.

```

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

```

```

/**
 * pre: -
 * @param posicion: valor entre 1 y el largo del vector
 * @throws Exception: da error si la posicion no esta en rango
 * post: remueve el valor en la posicion y deja el valor inicial
 */
public void remover(int posicion) throws Exception {
    if ((posicion < 1) ||
        (posicion > this.getLongitud())) {
        throw new Exception("La " + posicion + " no esta en el rango 1 y " + this.getLongitud() + " ir
    }
    this.datos[posicion - 1] = this.datoInicial;
}

```

```

10
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= public 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     int posicion = this.getLongitud();
91     this.datos = temp;
92     this.datos[posicion] = dato;
93     for(int i = posicion + 1; i < this.getLongitud(); i++) {
94         this.datos[i] = this.datoInicial;
95     }
96     return posicion + 1;
97 }
98

```

```

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

```

Listas

Definición:

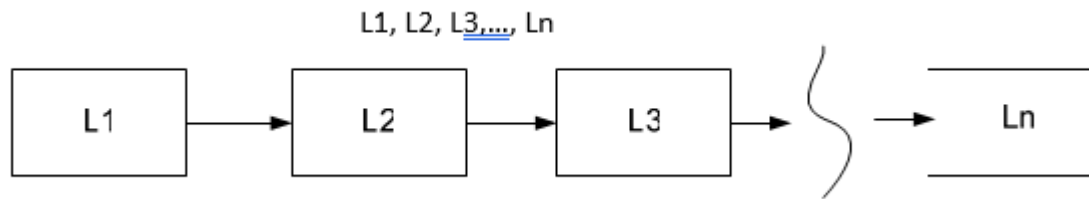
Una lista es una estructura de datos lineal flexible, ya que puede crecer (a medida que se insertan nuevos elementos) o acortarse (a medida que se borran elementos) según las necesidades que se presenten.

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

$L1, L2, \dots, L_n$

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

$x, L1, L2, \dots, L_n$

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.

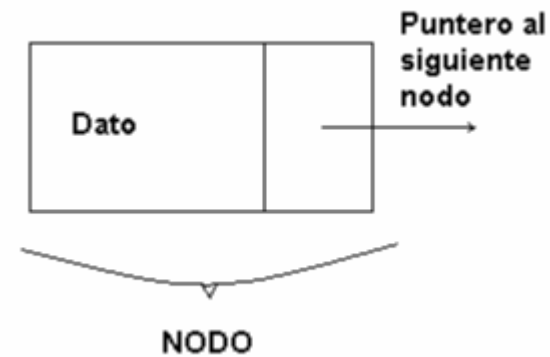
Implementacion de Lista con estructuras dinámicas

Lista simplemente enlazada

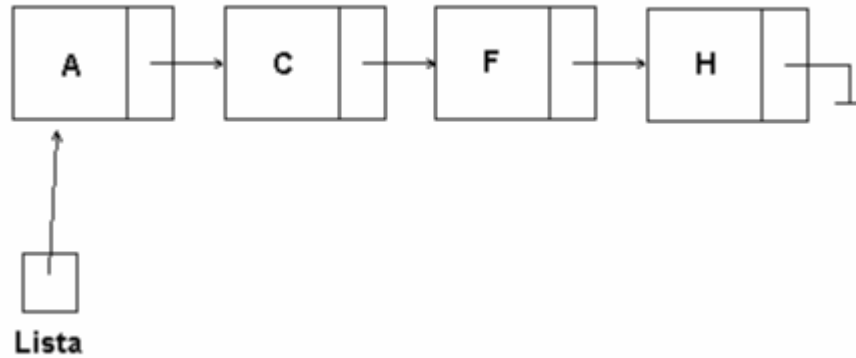
Antes de hablar de listas dinámicas debemos hablar de nodos. Las listas enlazarán nodos. Un nodo es una estructura que tendrá los siguientes datos:

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

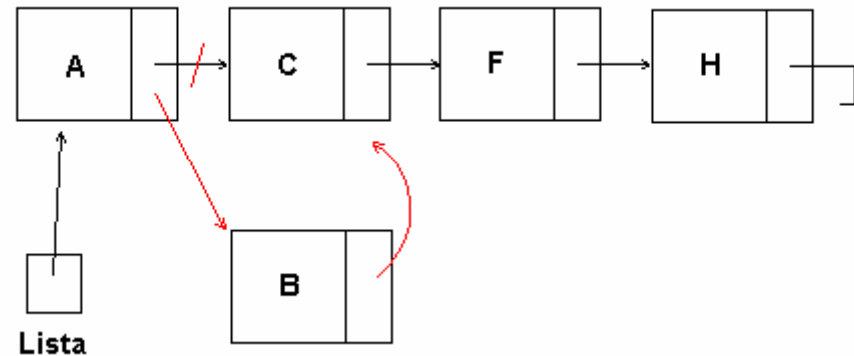
En el caso más simple, el nodo tendrá el elemento y un puntero al siguiente nodo de la lista:



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'.

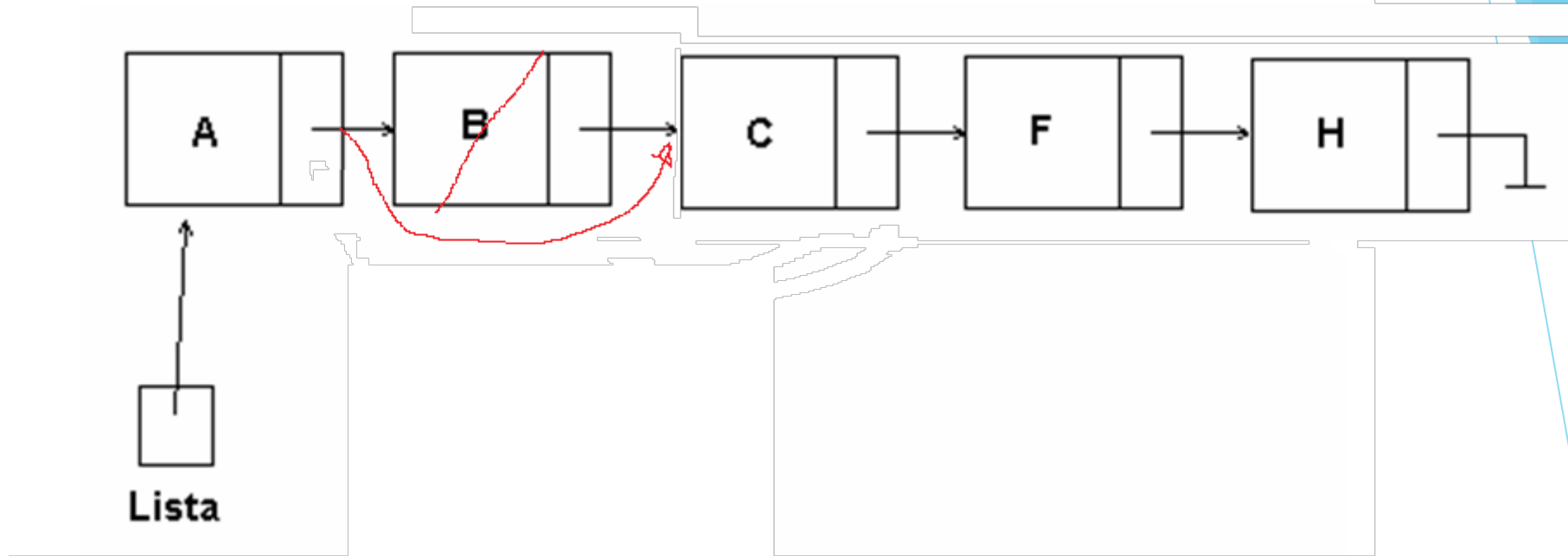
Estos nodos se irán creando en tiempo de ejecución, en forma dinámica, a medida que se vayan necesitando.

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_estática.

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 -----

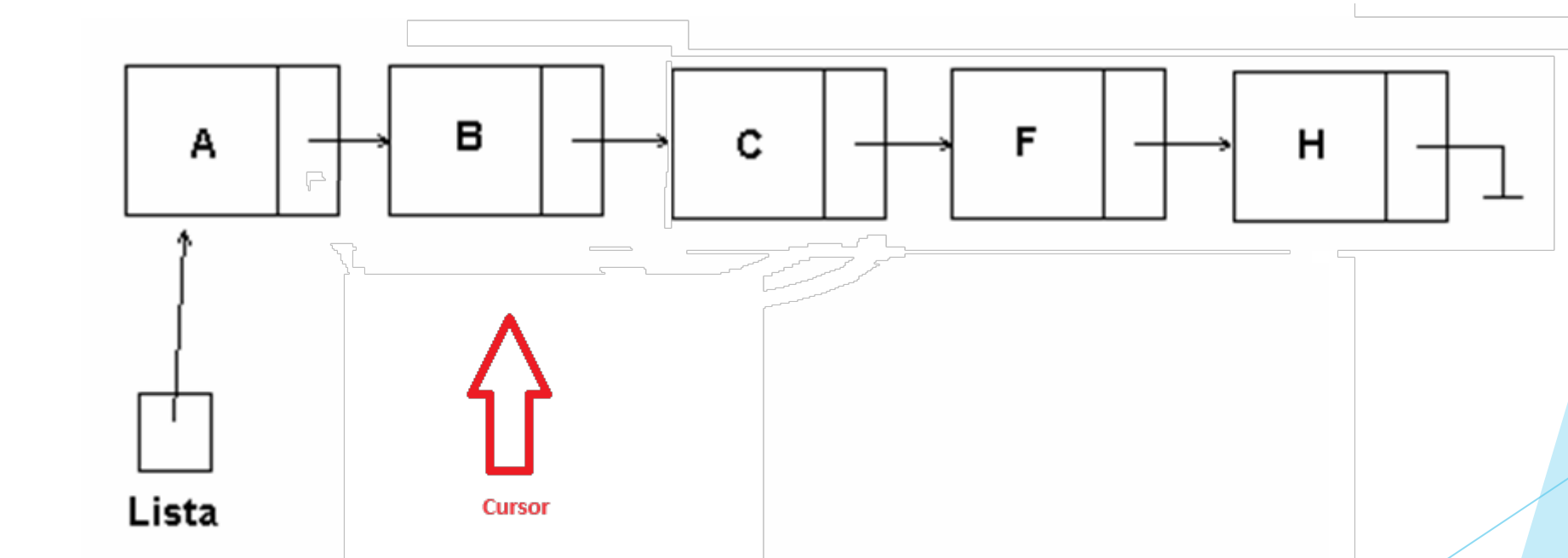
void setDato(int elemento, int posicion) throws Exception {
    validarPosicion(posicion);
    this.obtenerNodo(posicion).setDato(elemento);
}

```

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 con iterador e implementa la interfaz lista

¿Qué es una interfaz en Java?

Una interfaz en Java es una especie de contrato que define qué métodos debe tener una clase, pero no cómo los implementa. Es una estructura que contiene:

Métodos sin cuerpo (abstractos)

Constantes (campos public static final)

Desde Java 8: métodos default y static con implementación





```
public interface Vehiculo {  
    void arrancar();  
    void detener();  
}
```

```
public class Auto implements Vehiculo {  
    public void arrancar() {  
        System.out.println("Auto en marcha");  
    }  
  
    public void detener() {  
        System.out.println("Auto detenido");  
    }  
}
```

¿Qué es la interfaz List?

List es una interfaz de la colección de Java que representa una colección ordenada de elementos (es decir, una secuencia). Forma parte del framework de colecciones de Java y está definida en el paquete `java.util`.

✖ Características principales de List

-  **Ordenada:** mantiene el orden de inserción.
-  **Permite elementos duplicados:** a diferencia de `Set`, podés tener elementos repetidos.
-  **Acceso por índice:** podés acceder, insertar, eliminar elementos por su posición.
-  **Iteración:** se puede recorrer con `for`, `Iterator`, `ListIterator` y `Stream`.

```
Outline X
List<E>
  size() : int
  isEmpty() : boolean
  contains(Object) : boolean
  iterator() : Iterator<E>
  toArray() : Object[]
  toArray(T[]) <T> : T[]
  add(E) : boolean
  remove(Object) : boolean
  containsAll(Collection<?>) : boolean
  addAll(Collection<? extends E>) : boolean
  addAll(int, Collection<? extends E>) : boolean
  removeAll(Collection<?>) : boolean
  retainAll(Collection<?>) : boolean
  replaceAll(UnaryOperator<E>) : void
  sort(Comparator<? super E>) : void
  clear() : void
  equals(Object) : boolean
  hashCode() : int
  get(int) : E
  set(int, E) : E
  add(int, E) : void
  remove(int) : E
  indexOf(Object) : int
  lastIndexOf(Object) : int
  listIterator() : ListIterator<E>
  listIterator(int) : ListIterator<E>
  subList(int, int) : List<E>
  spliterator() : Spliterator<E>
  addFirst(E) : void
  addLast(E) : void
  getFirst() : E
  getLast() : E
  removeFirst() : E
  removeLast() : E
  reversed() : List<E>
  of() <E> : List<E>
```

```

9 public class ListaSimplementeEnlazada<T> implements List<T> {
10 //INTERFACES -----
11 //ENUMERADOS -----
12 //CONSTANTES -----
13 //ATRIBUTOS DE CLASE -----
14 //ATRIBUTOS -----
15
16     private NodoSimplementeEnlazado<T> primero = null;
17     private int tamaño = 0;
18     private NodoSimplementeEnlazado<T> cursor = null;
19
20 //ATRIBUTOS TRANSITORIOS -----
21 //CONSTRUCTORES -----
22
23     /**
24      * pre:
25      * pos: crea una lista vacia
26      */
27     public ListaSimplementeEnlazada() {}
28
29 //METODOS ABSTRACTOS -----
30 //METODOS HEREDADOS (CLASE)-----
31 //METODOS HEREDADOS (INTERFACE)-----
32 //METODOS DE CLASE -----
33 //METODOS GENERALES -----
34 //METODOS DE COMPORTAMIENTO -----
35

```



```

35
36- /**
37  * post: deja el cursor de la Lista preparado para hacer un nuevo
38  * recorrido sobre sus elementos.
39  */
40- public void iniciarCursor() {
41     this.cursor = null;
42 }
43
44- /**
45  * pre : se ha iniciado un recorrido (invocando el método
46  * iniciarCursor()) y desde entonces no se han agregado o
47  * removido elementos de la Lista.
48  * post: mueve el cursor y lo posiciona en el siguiente elemento
49  * del recorrido.
50  * El valor de retorno indica si el cursor quedó posicionado
51  * sobre un elemento o no (en caso de que la Lista esté vacía o
52  * no existan más elementos por recorrer.)
53  */
54- public boolean avanzarCursor() {
55     if (this.cursor == null) {
56         this.cursor = this.primerO;
57     } else {
58         this.cursor = this.cursor.getSiguiente();
59     }
60
61     /* pudo avanzar si el cursor ahora apunta a un nodo */
62     return (this.cursor != null);
63 }
64

```

```

65  /**
66   * pre : el cursor está posicionado sobre un elemento de la Lista,
67   *       (fue invocado el método avanzarCursor() y devolvió true)
68   * post: devuelve el elemento en la posición del cursor.
69   *
70   */
71  public T obtenerCursor() {
72      T elemento = null;
73      if (this.cursor != null) {
74          elemento = this.cursor.getDato();
75      }
76      return elemento;
77  }
78
79  /**
80   * pre: -
81   * pos: agrega el elemento al final de la Lista, en la posición:
82   *       contarElementos() + 1.
83   */
84  @Override
85  public boolean add(T elemento) {
86      addLast(elemento);
87      return true;
88  }
89

```

```

90= /**
91  * pre: -
92  * pos: agrega el elemento al final de la Lista, en la posición:
93  *       contarElementos() + 1.
94  */
95= public void addLast(T elemento) {
96     NodoSimplementeEnlazado<T> nuevo = new NodoSimplementeEnlazado<>(elemento);
97     if (primero == null) {
98         primero = nuevo;
99     } else {
100         NodoSimplementeEnlazado<T> actual = primero;
101         while (actual.tieneSiguiete()) {
102             actual = actual.getSiguiete();
103         }
104         actual.setSiguiete(nuevo);
105     }
106     tamano++;
107 }
108

```

```

109= /**
110  * pre: -
111  * pos: agrega el elemento en una posicion de la Lista, en la posición:
112  *       index, en el rango de 1 a n
113  */
114
115= @Override
116= public void add(int index, T elemento) {
117     if (index < 0 || index > tamano) {
118         throw new IndexOutOfBoundsException();
119     }
120
121     NodoSimplementeEnlazado<T> nuevo = new NodoSimplementeEnlazado<>(elemento);
122     if (index == 0) {
123         nuevo.setSiguiete(primero);
124         primero = nuevo;
125     } else {
126         NodoSimplementeEnlazado<T> actual = primero;
127         for (int i = 0; i < index - 1; i++) {
128             actual = actual.getSiguiete();
129         }
130         nuevo.setSiguiete(actual.getSiguiete());
131         actual.setSiguiete(nuevo);
132     }
133     tamano++;
134 }
135

```

```

136- /**
137-  * pre:
138-  * pos: indica si la Lista tiene algún elemento.
139-  */
140- @Override
141- public boolean isEmpty() {
142-     return tamaño == 0;
143- }
144-
145- /**
146-  * Devuelve el elemento de una posición de la lista
147-  */
148- @Override
149- public T get(int index) {
150-     if (index < 0 || index >= tamaño) throw new IndexOutOfBoundsException();
151-     NodoSimplementeEnlazado<T> actual = primero;
152-     for (int i = 0; i < index; i++) {
153-         actual = actual.getSiguiente();
154-     }
155-     return actual.getDato();
156- }
157-

```

```

157-
158- /**
159-  * Demueve el elemento de una posición de la lista
160-  */
161- @Override
162- public T remove(int index) {
163-     if (index < 0 || index >= tamaño) throw new IndexOutOfBoundsException();
164-     NodoSimplementeEnlazado<T> eliminado;
165-     if (index == 0) {
166-         eliminado = primero;
167-         primero = primero.getSiguiente();
168-     } else {
169-         NodoSimplementeEnlazado<T> actual = primero;
170-         for (int i = 0; i < index - 1; i++) {
171-             actual = actual.getSiguiente();
172-         }
173-         eliminado = actual.getSiguiente();
174-         actual.setSiguiente( eliminado.getSiguiente());
175-     }
176-     tamaño--;
177-     return eliminado.getDato();
178- }
179-

```

```

180  /**
181   * Devuelve el tamaño
182   */
183  @Override
184  public int size() {
185      return tamaño;
186  }
187
188  /**
189   * Crea un iterador para recorrer la lista
190   */
191  @Override
192  public Iterator<T> iterator() {
193      return new Iterator<T>() {
194          private NodoSimplementeEnlazado<T> actual = primero;
195
196          public boolean hasNext() {
197              return actual != null;
198          }
199
200          public T next() {
201              T dato = actual.getDato();
202              actual = actual.getSiguiente();
203              return dato;
204          }
205      };
206  }
207

```

```

211  @Override
212  public boolean contains(Object o) {
213      for (T elemento : this) {
214          if (Objects.equals(elemento, o)) return true;
215      }
216      return false;
217  }
218
219  /**
220   * Elimina un elemento de la lista y sino lo encuentra devuelve falso
221   */
222  @Override
223  public boolean remove(Object o) {
224      if (primero == null) return false;
225
226      if (Objects.equals(primero.getDato(), o)) {
227          primero = primero.getSiguiente();
228          tamaño--;
229          return true;
230      }
231
232      NodoSimplementeEnlazado<T> actual = primero;
233      while (actual.tieneSiguiente() &&
234             !Objects.equals(actual.getSiguiente().getDato(), o)) {
235          actual = actual.getSiguiente();
236      }
237
238      if (actual.getSiguiente() != null) {
239          actual.setSiguiente(actual.getSiguiente().getSiguiente());
240          tamaño--;
241          return true;
242      }
243
244      return false;
245  }
246

```

```

240  /**
241  * Elimina todos los elementos de la lista
242  */
243  @Override
244  public void clear() {
245      primero = null;
246      tamaño = 0;
247  }
248
249  /**
250  * cambia el elemento de una posición
251  */
252  @Override
253  public T set(int index, T element) {
254      if (index < 0 || index >= tamaño) throw new IndexOutOfBoundsException();
255      NodoSimplementeEnlazado<T> actual = primero;
256      for (int i = 0; i < index; i++) {
257          actual = actual.getSiguiente();
258      }
259      T viejo = actual.getDato();
260      actual.setDato(element);
261      return viejo;
262  }
263
264

```

```

270
271  /**
272  * Devuelve el índice de un item en la lista
273  */
274  @Override
275  public int indexOf(Object o) {
276      int index = 0;
277      for (T elemento : this) {
278          if (Objects.equals(elemento, o)) return index;
279          index++;
280      }
281      return -1;
282  }
283
284  /**
285  * Devuelve el último índice de un elemento en la lista
286  */
287  @Override
288  public int lastIndexOf(Object o) {
289      int index = 0, ultimo = -1;
290      for (T elemento : this) {
291          if (Objects.equals(elemento, o)) ultimo = index;
292          index++;
293      }
294      return ultimo;
295  }
296

```

```

297
298 @Override
299 public Object[] toArray() {
300     Object[] array = new Object[tamano];
301     int i = 0;
302     for (T elem : this) {
303         array[i++] = elem;
304     }
305     return array;
306 }
307
308 @SuppressWarnings("unchecked")
309 @Override
310 public <E> E[] toArray(E[] a) {
311     if (a.length < tamano) {
312         a = (E[]) java.lang.reflect.Array.newInstance(a.getClass().getComponentType(), tamano);
313     }
314
315     int i = 0;
316     for (T elem : this) {
317         a[i++] = (E) elem;
318     }
319
320     if (a.length > tamano) {
321         a[tamano] = null;
322     }
323
324     return a;
325 }

```

```

326
327 @Override
328 public boolean containsAll(Collection<?> c) {
329     for (Object elem : c) {
330         if (!contains(elem)) return false;
331     }
332     return true;
333 }
334
335 @Override
336 public boolean addAll(Collection<? extends T> c) {
337     boolean cambiado = false;
338     for (T elem : c) {
339         add(elem);
340         cambiado = true;
341     }
342     return cambiado;
343 }
344
345 @Override
346 public boolean addAll(int index, Collection<? extends T> c) {
347     if (index < 0 || index > tamaño) throw new IndexOutOfBoundsException();
348
349     boolean modificado = false;
350     for (T elem : c) {
351         add(index++, elem);
352         modificado = true;
353     }
354     return modificado;
355 }
356

```



```

357 @Override
358 public boolean removeAll(Collection<?> c) {
359     boolean modificado = false;
360     for (Object elem : c) {
361         while (remove(elem)) {
362             modificado = true;
363         }
364     }
365     return modificado;
366 }
367
368 @Override
369 public boolean retainAll(Collection<?> c) {
370     boolean modificado = false;
371     Iterator<T> it = this.iterator();
372     while (it.hasNext()) {
373         if (!c.contains(it.next())) {
374             it.remove();
375             modificado = true;
376         }
377     }
378     return modificado;
379 }
380
381 @Override
382 public ListIterator<T> listIterator() {
383     throw new UnsupportedOperationException("listIterator() no está implementado aún");
384 }
385
386 @Override
387 public ListIterator<T> listIterator(int index) {
388     throw new UnsupportedOperationException("listIterator(int) no está implementado aún");
389 }

```

```

390
391 @Override
392 public List<T> subList(int fromIndex, int toIndex) {
393     if (fromIndex < 0 || toIndex > tamaño || fromIndex > toIndex)
394         throw new IndexOutOfBoundsException();
395
396     ListaSimplementeEnlazada<T> sublista = new ListaSimplementeEnlazada<>();
397     NodoSimplementeEnlazado<T> actual = primero;
398     for (int i = 0; i < toIndex; i++) {
399         if (i >= fromIndex) sublista.add(actual.getDato());
400         actual = actual.getSiguiente();
401     }
402     return sublista;
403 }
404
405 //METODOS DE CONSULTA DE ESTADO -----
406 //GETTERS REDEFINIDOS -----
407 //GETTERS INICIALIZADOS -----
408 //GETTERS COMPLEJOS -----
409 //GETTERS SIMPLES -----
410 //SETTERS COMPLEJOS-----
411 //SETTERS SIMPLES -----
412
413 }

```

Recorridos


Comparación de métodos de recorrido

Método	Ventajas	Desventajas	Ideal para...
<code>for</code> tradicional	Acceso por índice, control total	Verboso, poco eficiente en listas enlazadas	Acceso aleatorio (<code>ArrayList</code>)
<code>for-each</code> (: <code>for</code>)	Claro y legible	No permite modificar la lista	Recorridos simples de lectura
<code>Iterator</code>	Permite eliminar elementos	Más código, sin acceso al índice	Recorridos donde se eliminan elementos
<code>ListIterator</code>	Puede recorrer en ambos sentidos, insertar/modificar	Solo <code>List</code> lo soporta, más complejo	Modificación avanzada durante el recorrido
<code>Stream</code>	Sintaxis funcional y poderosa	No permite modificar la colección	Procesamiento, filtrado y transformación de datos

1. for tradicional

java


 Copiar


 Editar

```
for (int i = 0; i < lista.size(); i++) {  
    System.out.println(lista.get(i));  
}
```

2. for-each

java

 Copiar


 Editar

```
for (String s : lista) {  
    System.out.println(s);  
}
```

3. Iterator

java

 Copiar


 Editar

```
Iterator<String> it = lista.iterator();  
while (it.hasNext()) {  
    String s = it.next();  
    System.out.println(s);  
}
```

4. ListIterator (avance y retroceso)

java


 Copiar


 Editar

```
ListIterator<String> it = lista.listIterator();  
while (it.hasNext()) {  
    System.out.println("Adelante: " + it.next());  
}  
while (it.hasPrevious()) {  
    System.out.println("Atrás: " + it.previous());  
}
```

5. Stream

java

 Copiar

 Editar

```
lista.stream().forEach(System.out::println);
```

```
public class EjemploListIterator {
    public static void main(String[] args) {
        List<String> lista = new ArrayList<>(Arrays.asList("A", "B", "C"));
        ListIterator<String> it = lista.listIterator();

        System.out.println("Recorrido hacia adelante:");
        while (it.hasNext()) {
            String actual = it.next();
            System.out.println("Elemento: " + actual);

            // Modificar elementos
            if (actual.equals("B")) {
                it.set("Beta");
            }

            // Agregar un nuevo elemento después de "A"
            if (actual.equals("A")) {
                it.add("X");
            }
        }

        System.out.println("\nRecorrido hacia atrás:");
        while (it.hasPrevious()) {
            System.out.println("Elemento: " + it.previous());
        }

        System.out.println("\nLista final: " + lista);
    }
}
```

```
import java.util.*;
import java.util.stream.Collectors;

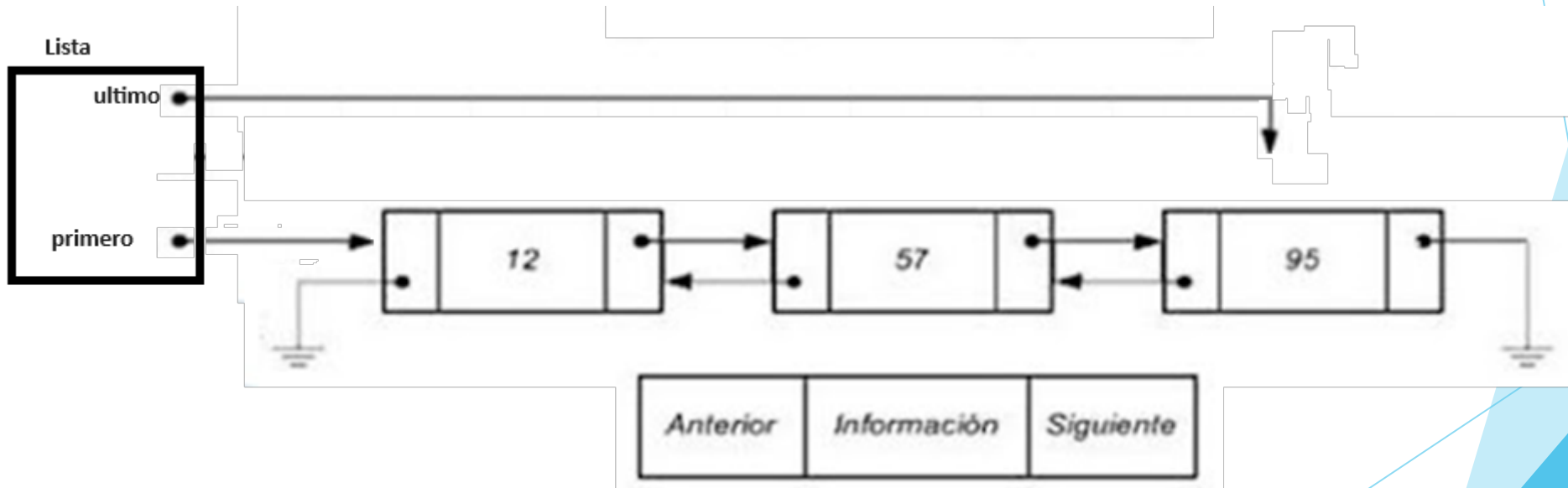
public class EjemploStream {
    public static void main(String[] args) {
        List<String> lista = Arrays.asList("A", "B", "C", "B");

        // Filtrar, transformar y recolectar en nueva lista
        List<String> resultado = lista.stream()
            .filter(s -> !s.equals("B"))    // Eliminar "B"
            .map(String::toLowerCase)      // Transformar a minúsculas
            .collect(Collectors.toList());

        System.out.println("Lista original: " + lista);
        System.out.println("Lista transformada: " + resultado);
    }
}
```

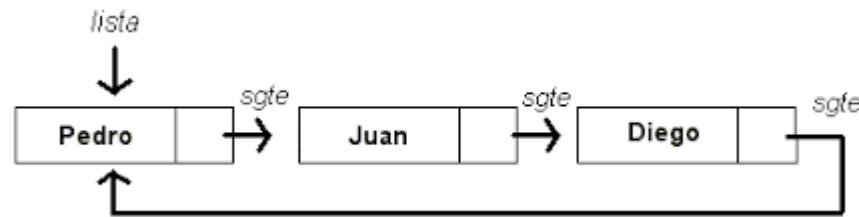
Lista doblemente enlazada

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.

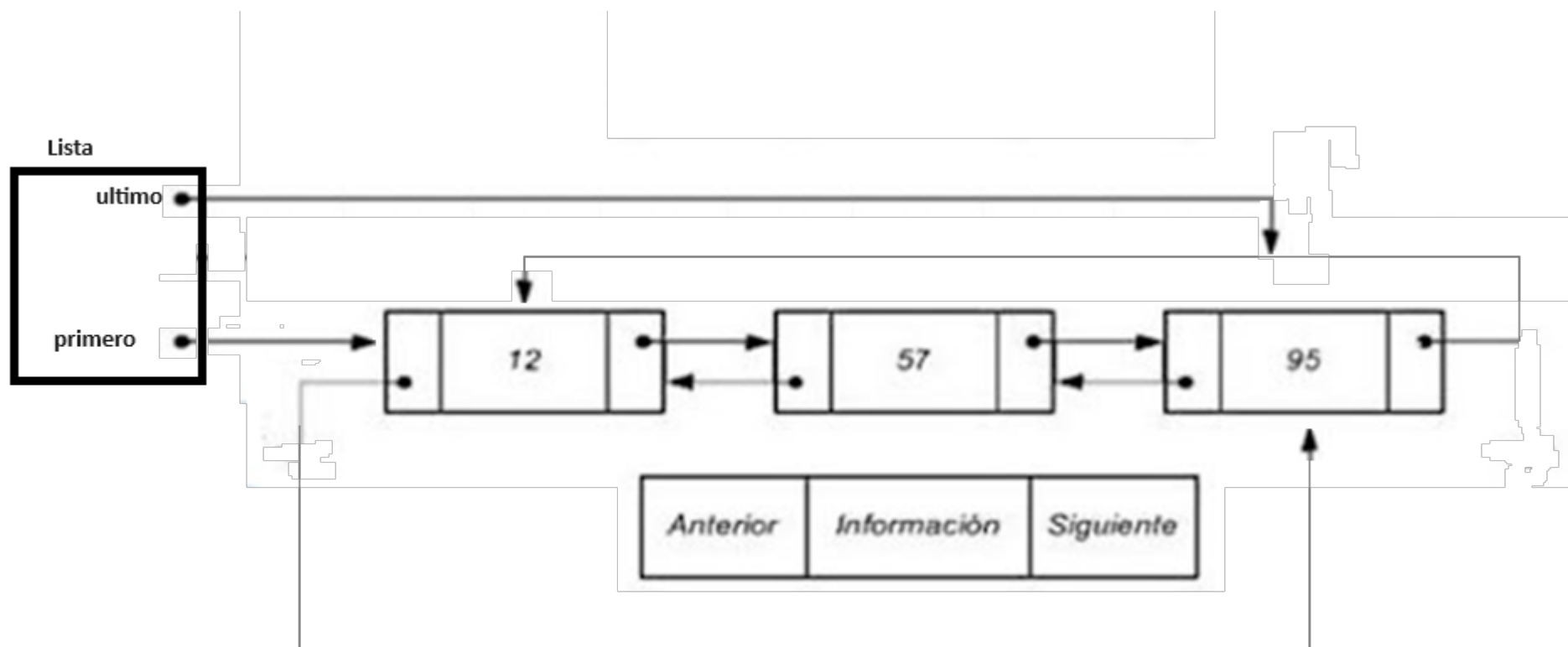


Lista circular

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.

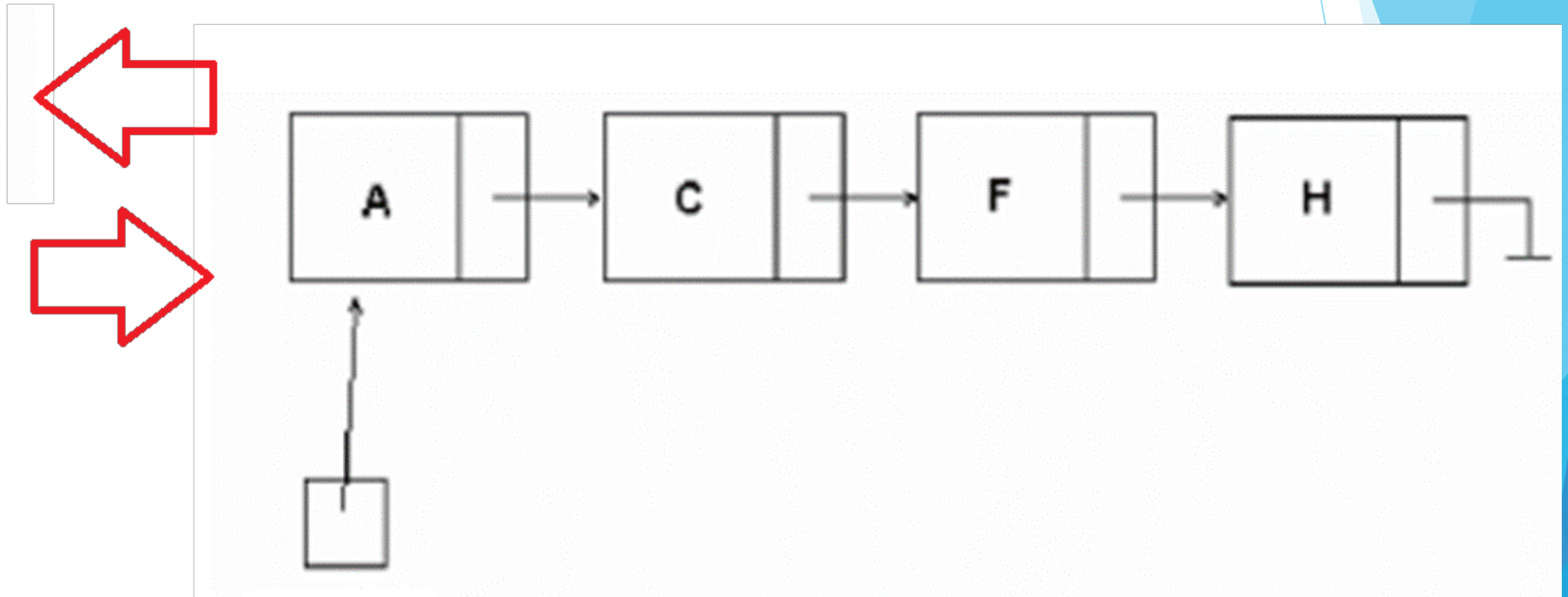
En ese sentido puede decirse que una pila es una clase especial de lista en la cual todas las inserciones (alta ,apilar o push) y borrados (baja, desapilar o pop) tienen lugar en un extremo denominado cabeza o tope.

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').

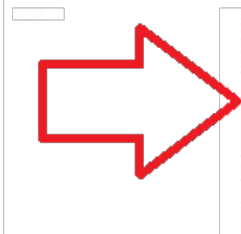
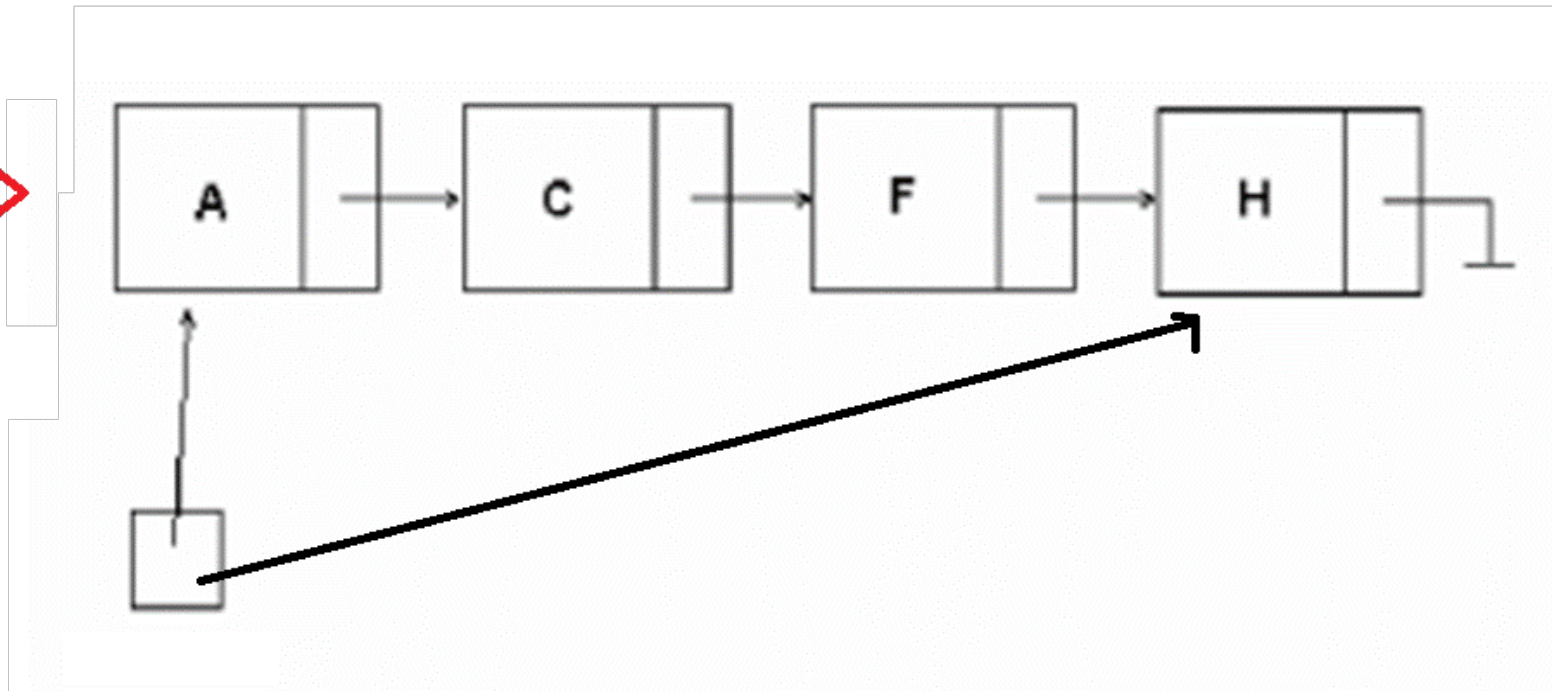
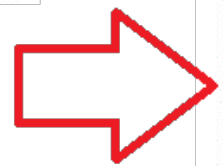
En ese sentido puede decirse que una cola es una lista con restricciones en el alta (encolar o enqueue) y baja (desencolar, dequeue).

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<Solucion*>* buscarSolucionesEquivalente(Solucion* solucionRequerida, Lista<Solucion*>* 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<Compuesto*>* 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