

# Estructuras de datos lineales

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

# Templates en C++

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.

## Ejemplo de función sin template

```
void swap (int& a, int& b){  
    int c=a;  
    a=b;  
    b=c;  
}
```

```
void swap (string & a, string & b) {  
    string c=a;  
    a=b;  
    b=c;  
}
```

Si tuviéramos 20 tipos de datos, tendríamos que implementar 20 funciones distintas.

Versión de la función con template:

```
template <class T>
void swap (T& a, T &b) {
    T c=a;
    a=b;
    b=c;
}
```

```
int main() {
    int a = 1, b = 2;
    swap <int>(a, b);
    return 0;
}
```

¿class o typename?

En las primeras versiones de c++, sólo existía class. Luego typename se incorpora posteriormente. Es una palabra reservada **imprescindible** para resolver ciertos **conflictos** que pueden darse cuando se ha definido una clase anidada en otra.

En las situaciones que veremos en la cursada, son igualmente válidas typename y class.

```
template <class T>
void swap (T a, T b) {
    T c=a;
    a=b;
    b=c;
}
```

```
template <typename T>
void swap (T a, T b) {
    T c=a;
    a=b;
    b=c;
}
```

## Template en TDA

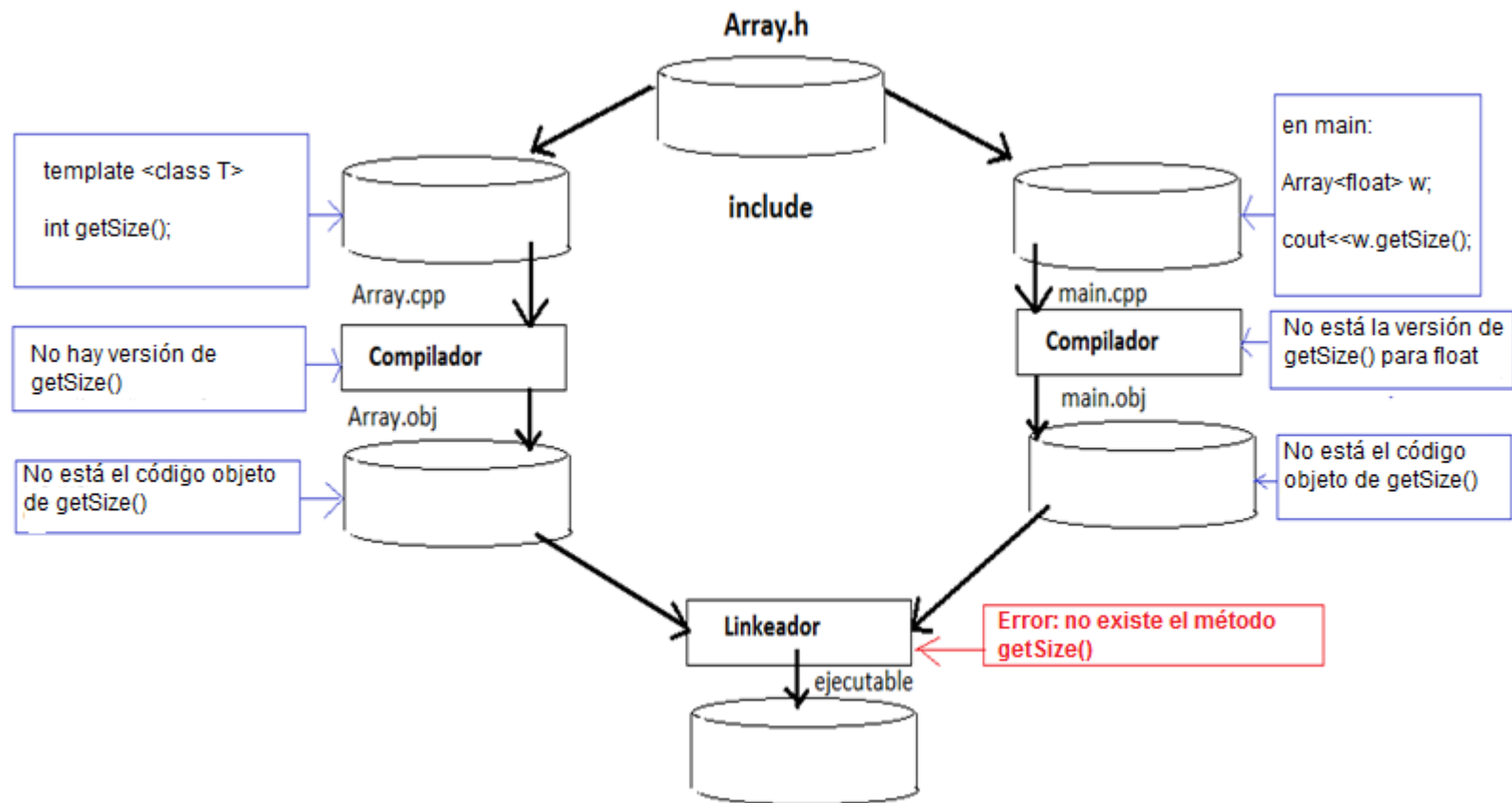
```
2⊕ * Plantilla.h
7
8 #ifndef PLANTILLA H
9 #define PLANTILLA H
10
11⊖ template <class T>
12 class Plantilla {
13 private:
14     T dato;
15
16 public:
17     Plantilla() {}
18     virtual ~Plantilla() {}
19
20⊖     T getData() {
21         return this->dato;
22     }
23
24⊖     void setData(T dato) {
25         this->dato = dato;
26     }
27 };
28
29 #endif /* PLANTILLA H */

17 ,
18
19⊖ int main() {
20     Plantilla<int> * plantillaDeEntero = new Plantilla<int>();
21     plantillaDeEntero->setDato(9);
22     std::cout << plantillaDeEntero->getData();
23
24 }
```

¿Qué hubiera pasado de haber planteado la clase template de TDA con dos archivos, por ejemplo un TDA Array, .h y .cpp?

Tendríamos Array.h y Array.cpp, y la función main.

Ejemplificamos el problema con el método getSize()



## Plantilla con mas de un tipo de dato

También se puede hacer una plantilla para mas de un tipo de datos, se pueden agregar todos los tipos necesarios:

```
// C++ Program to implement
// Use of template
#include <iostream>
using namespace std;

template <class T, class U> class A {
    T x;
    U y;

public:
    A() { cout << "Constructor Called" << endl; }
};

int main()
{
    A<char, char> a;
    A<int, double> b;
    return 0;
}
```

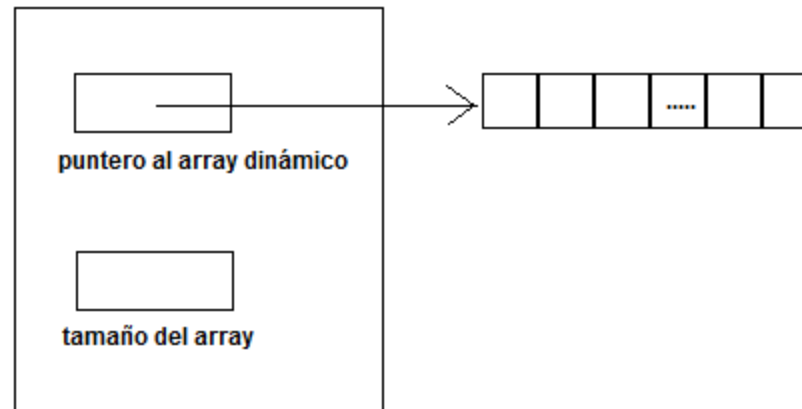


# Estructuras de Datos lineales

- ▶ Vector
- ▶ Vector con 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:

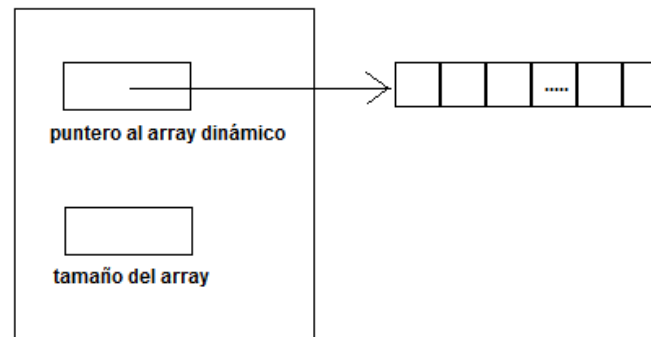
Las primitivas deben hacer lo siguiente:

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



```

1  #ifndef VECTOR_H_
2  #define VECTOR_H_
3
4  template<class T> class Vector {
5  private:
6      T * datos;
7      int longitud;
8
9  public:
10
11      /**
12       * pre: la longitud es mayor o igual a 1. Y el dato inicial puede ser NULL si es un puntero
13       * pos: deja un vector con las posiciones solicitadas
14       */
15      Vector(int longitud, T datoInicial) {
16          if (longitud < 1) {
17              throw "La longitud debe ser mayor o igual a 1";
18          }
19          this->datos = new T[longitud];
20          this->longitud = longitud;
21          for(int i = 0; i < this->longitud; i++){
22              this->datos[i] = datoInicial;
23          }
24      }
25
26      /**
27       * pre:
28       * post: libera la memoria
29       */
30      virtual ~Vector() {
31          delete [] this->datos;
32      }
33
34      /**
35       * pre:
36       * post: devuelve la longitud actual del vector
37       */
38      int getLongitud() {
39          return this->longitud;
40      }
41
42      /**
43       * pre: la posicion esta entre 1 y n (inclusive)
44       * pos: guarda el dato en la posicion indicada, sino devuelve error
45       */
46      void agregar(int posicion, T dato) {
47          if ((posicion < 1) ||
48              (posicion > this->longitud)) {
49              throw "La " + posicion + " no esta en el rango 1 y " + this->longitud + " inclusive";
50          }
51          this->datos[posicion - 1] = dato;
52      }
53
54      /**
55       * pre: la posicion esta entre 1 y n (inclusive)
56       * pos: guarda el dato en la posicion indicada, sino devuelve error
57       */
58      T& obtener(int posicion) {
59          if ((posicion < 1) ||
60              (posicion > this->longitud)) {
61              throw "La " + posicion + " no esta en el rango 1 y " + this->longitud + " inclusive";
62          }
63          return this->datos[posicion - 1];
64      }
65  };
66
67  #endif /* VECTOR_H_ */

```

## 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: -
 * pos: agregar el dato en la primer posicion vacia. Sino hay mas espacio,
 *      agranda el vector al doble del tamaño.
 */
int agregar(T dato) {
    for(int i = 0; i < this->longitud; i++){
        if (this->datos[i] == datoInicial) {
            this->datos[i] = dato;
            return i + 1;
        }
    }
    T * temp = new T[this->longitud * 2];
    for(int i = 0; i < this->longitud; i++){
        temp[i] = this->datos[i];
    }
    delete [] this->datos;
    this->datos = temp;
    this->datos[this->longitud + 1] = dato;
    this->longitud *= 2;
    return this->longitud / 2 + 1;
}
```

# Listas

## Definición:

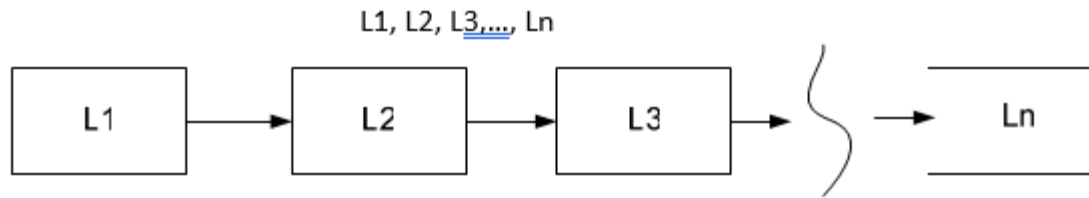
Una lista es una estructura de datos lineal flexible, ya que puede crecer (a medida que se insertan nuevos elementos) o acortarse (a medida que se borran elementos) según las necesidades que se presenten.

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.

## Distintas formas de implementación

¿Cómo se lleva a la práctica toda la teoría? Básicamente, podremos dividir la implementación en dos grupos: estructuras estáticas y estructuras dinámicas.

Las estructuras estáticas: utiliza un bloque de memoria contiguo (solo demostrativo)

Las estructuras dinámicas: no utilizan un bloque de memoria contiguo, sino mucho bloques mínimos independientes.

## Estructuras estáticas

Una implementación sencilla es con un vector de elementos (pueden ser tipos simples, structs - registros - u otros objetos), donde cada posición del vector contiene un elemento. Este vector se define en forma estática, es decir, su tamaño debe ser definido en tiempo de compilación.

Por ejemplo, una lista cuyos elementos sean caracteres, podría verse de la siguiente manera:

<u>max_length</u>		6
		5
last	'F'	4
	'C'	3
	'H'	2
first	'A'	1

En este ejemplo se definió un vector de 6 posiciones (`max_length = 6`) y sus elementos son, en forma secuencial, 'A', 'H', 'C', 'F'. El tamaño de la lista es 4 (`last`) y, como se observa, no conserva un orden.

Nota: el índice que figura al costado significa que 'A' es el primer elemento, 'H' el segundo, etc. No es el índice del vector, ya que en C++ los índices comienzan en 0.

```
8 #ifndef LISTA_H INCLUDED
9 #define LISTA_H INCLUDED
10 // Tamaño máximo de la lista const int MAX TAM = 10;
11 static const int MAX TAM = 10;
12
13 /** Clase Lista estatica Implementada con un vector de elementos
14 y un tamaño fijo (MAX TAM) */
15
16 class ListaEstatica {
17 private:
18     // Vector donde se iran agregando los elementos dato lista[MAX TAM];
19     // Tamaño lógico de la lista int tope;
20     char datos[MAX TAM];
21     int tope;
22
23 public:
24     // Constructor
25     // PRE: ninguna
26     // POST: crea una lista vacía con tope = 0 Lista estatica();
27
```



## Algunas aclaraciones con respecto al ejemplo de implementación

- En primer lugar, el nombre de la clase `ListaEstatica`, se utiliza sólo con fines didácticos y para diferenciar de otras implementaciones que se harán más adelante. Pero no es recomendable utilizar nombres de clases que indiquen la forma de implementación. En este caso, lo correcto sería haber utilizado `Lista` a secas.

¿Por qué no debemos indicar el tipo de implementación en el nombre?

Por varias razones. Una de las razones, es que el paradigma de TDA nos habla de ocultamiento de la implementación y abstracción de datos. Es decir, utilizar algo, sabiendo cómo se utiliza pero sin preocuparnos cómo está hecho internamente. Por ejemplo, cuando ponemos un DVD en una reproductora y presionamos play, sabemos que podremos ver la película que colocamos, pero no nos interesa saber el mecanismo interno que hace la reproductora para reproducirla. Conocer el detalle interno de cada artefacto que utilizamos y estar pendientes del mismo nos confundiría y haría nuestra vida muy complicada. Por otro lado, nos veríamos tentados a “meter mano” dentro de los artefactos lo cual no sería conveniente. Por algo las garantías se invalidan cuando el usuario intenta arreglar el dispositivo por su propia cuenta.

Otro motivo es que el día de mañana podríamos querer cambiar nuestra implementación (que actualmente estamos utilizando) por otra, por ejemplo, por una lista dinámica. De esta forma tendríamos que cambiar en todo el código las indicaciones que digan `ListaEstatica` por `ListaDinamica`, lo cual no tiene sentido.

- Esta implementación es muy sencilla pero no es muy útil. No tendría sentido utilizar una lista para almacenar 10 caracteres. Si bien la constante `MAX_TAM` podría cambiarse fácilmente de 10 a 100 o 1000, esta implementación no se volvería demasiado provechosa ya que presenta otros inconvenientes que serán explicados. Si necesitáramos generalizar (poder albergar distintos tipos de datos), también podríamos, fácilmente, cambiar el tipo de dato de `char` a `int` o a `float`. Sin embargo, si quisiéramos utilizar elementos más complejos, esta implementación no serviría. Hay que tener en cuenta, lo que se comentaba en la sección 3, sobre el método

`obtener` (en este ejemplo es `getDato`) que debería devolver un puntero para que a) sea eficiente y b) pueda devolver un nulo cuando no ubica el objeto buscado.

- Con respecto a las posiciones pasadas por parámetro, tanto en el `obtener` como en el `eliminar`, no se verifican que los valores sean válidos porque esto es responsabilidad del usuario de la clase, ya que se indica en las pre condiciones correspondientes que la posición debe ser mayor a cero y menor o igual que el tope.

- En cuanto a la inserción, se realiza siempre al final de la lista. En muchas implementaciones, este método lo llaman `agregar`, debido a que agrega el elemento al final de la lista. Es una implementación muy simple de realizar pero, generalmente, no será la ideal, salvo en el caso de una pila o cola que veremos más adelante.

- Nótese que, como la implementación es estática y los datos que se guardan también lo son, no hay necesidad de liberar memoria, por lo cual, el destructor no realiza nada (se podría haber dejado el destructor de oficio que provee el lenguaje).

¿Qué problemas tiene una implementación estática?

I. El primer inconveniente que se presenta es el problema de estimar la dimensión antes de ejecutar el programa. Si estimamos muy poco corremos el riesgo de que se nos termine nuestro vector y no podamos seguir almacenando elementos. Por el contrario, si sobrestimamos, consumiremos memoria de más, lo cual puede llegar a ser crítico, ya que, además de poder quedarnos sin memoria disponible, la aplicación podría volverse notoriamente lenta.

Si bien podríamos definir un vector de forma dinámica, deberíamos pedirle al usuario que ingresara el tamaño necesitado al principio de la aplicación. Transfiriéndole a él el mismo problema que se acaba de señalar.

II. En segundo lugar, si observamos el método de borrado, vemos que, para borrar un elemento que se encuentra en la posición  $p$ , debemos “correr” un lugar todos los elementos que están en las posiciones  $p+1, p+2, \dots, \text{tope}$ , ubicándolos en las posiciones  $p, p+1, \dots, \text{tope}-1$ , respectivamente.

En el insertado, a excepción de la inserción al final (como se implementó), se debería realizar una operación similar pero a la inversa, para crear el lugar en donde iría el nuevo elemento.

En la siguiente figura se muestran, con flechas, los movimientos que se deberían realizar para pasar del Estado 1 al Estado 2 de una lista ordenada, al insertar la letra 'B'.

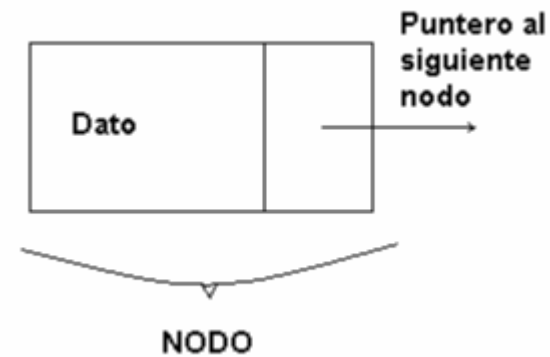
## Estructuras dinámicas

### Lista simplemente enlazada

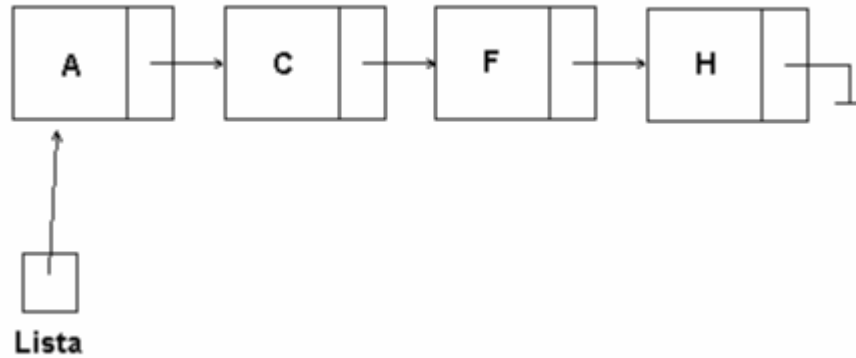
Antes de hablar de listas dinámicas debemos hablar de nodos. Las listas enlazarán nodos. Un nodo es una estructura que tendrá los siguientes datos:

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

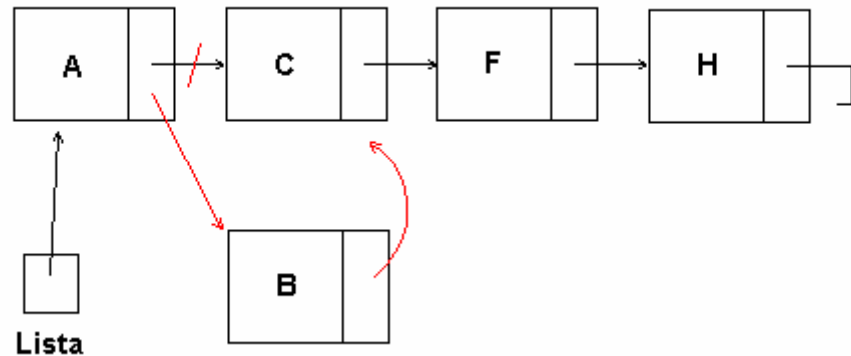
En el caso más simple, el nodo tendrá el elemento y un puntero al siguiente nodo de la lista:



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'.

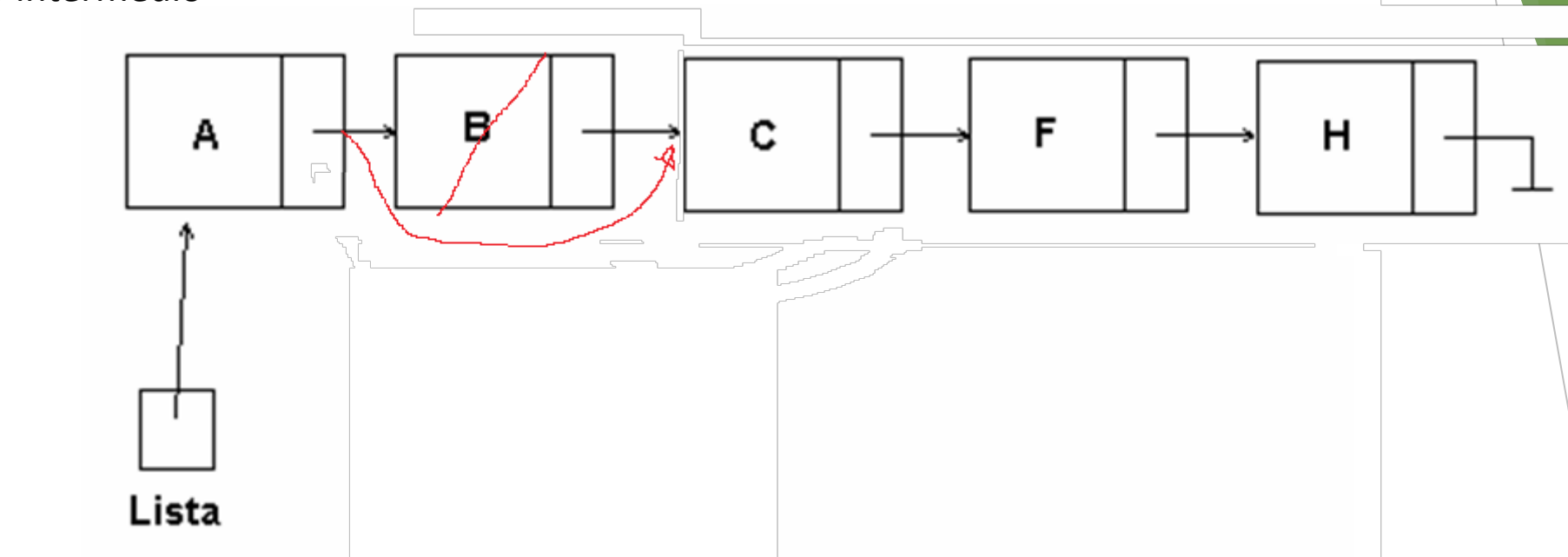
Estos nodos se irán creando en tiempo de ejecución, en forma dinámica, a medida que se vayan necesitando.

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.

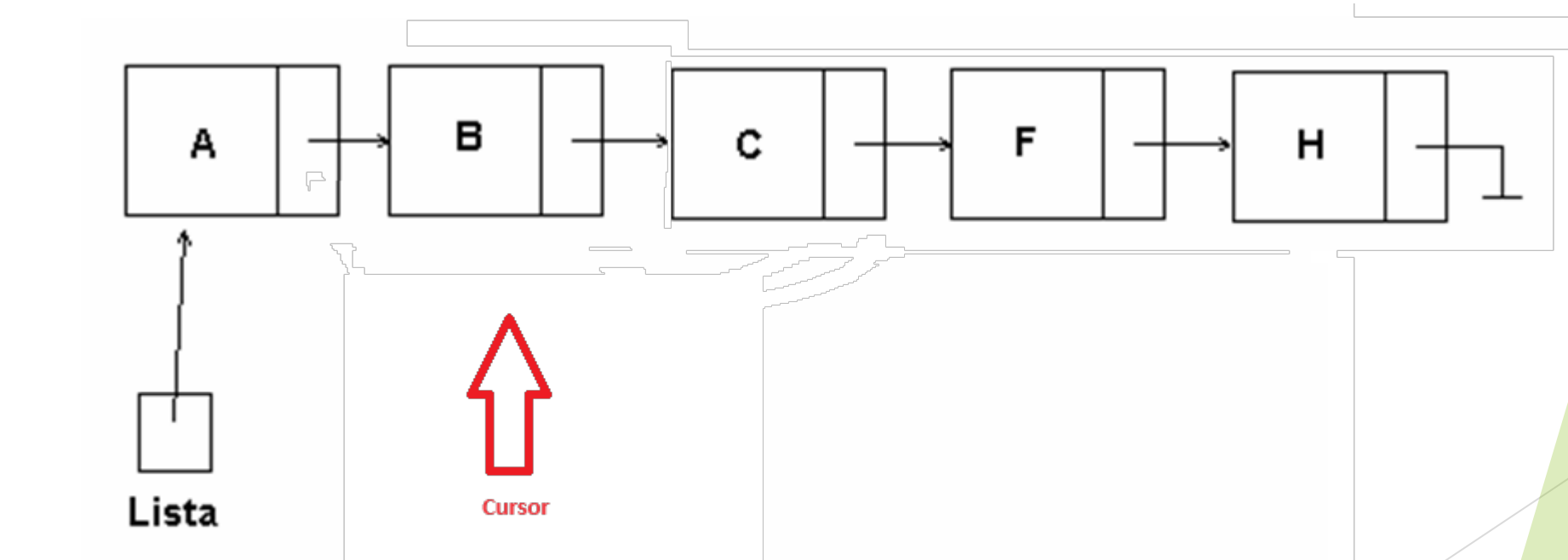
```
1  #ifndef LISTA_H_
2  #define LISTA_H_
3
4  #include "Nodo.h"
5
6  /*
7   * Una Lista es una colección dinámica de elementos dispuestos en una secuencia.
8   *
9   * Define operaciones para agregar, remover, acceder y cambiar elementos
10  * en cualquier posición.
11  *
12  * Tiene un cursor que permite recorrer todos los elementos secuencialmente.
13  *
14  */
15  ✓ template<class T> class Lista {
16
17      private:
18
19          Nodo<T>* primero;
20
21          unsigned int tamano;
22
23          
24
25      public:
26
27          /*
28           * post: Lista vacía.
29           */
30          Lista();
31
32          /*
33           * post: Lista que tiene los mismos elementos que otraLista.
34           *      La instancia resulta en una copia de otraLista.
35           */
```



## 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.



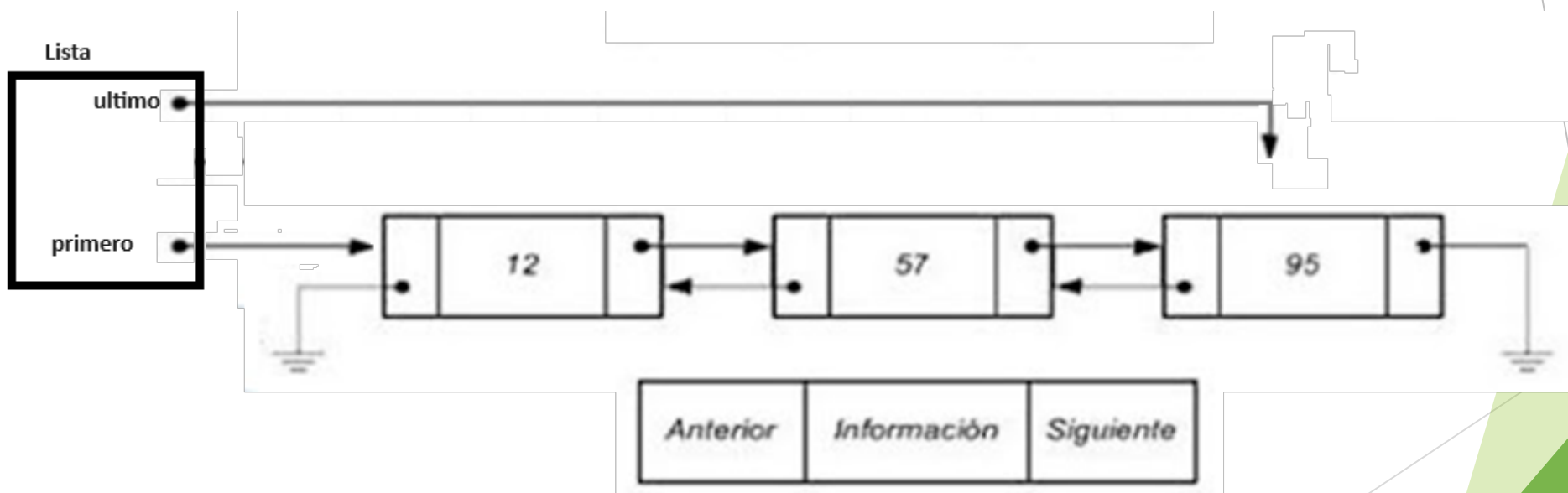
```
1  #ifndef LISTA_H_
2  #define LISTA_H_
3
4  #include "Nodo.h"
5
6  /*
7   * Una Lista es una colección dinámica de elementos dispuestos en una secuencia.
8   *
9   * Define operaciones para agregar, remover, acceder y cambiar elementos
10  * en cualquier posición.
11  *
12  * Tiene un cursor que permite recorrer todos los elementos secuencialmente.
13  *
14  */
15  ✓ template<class T> class Lista {
16
17      private:
18
19          Nodo<T>* primero;
20
21          unsigned int tamano;
22
23          Nodo<T>* cursor;
24
25      public:
26
27          /*
28           * post: Lista vacía.
29           */
30          Lista();
31
32          /*
33           * post: Lista que tiene los mismos elementos que otraLista.
34           *      La instancia resulta en una copia de otraLista.
35           */
```

## Recorrido de la lista

```
11
12 int main(int argc, char **argv) {
13
14
15     Lista<int> * otraLista = new Lista<int>();
16
17     //....
18     otraLista->iniciarCursor();
19     while (otraLista->avanzarCursor()) {
20         std::out << otraLista->obtenerCursor());
21     }
22
23 }
24
```

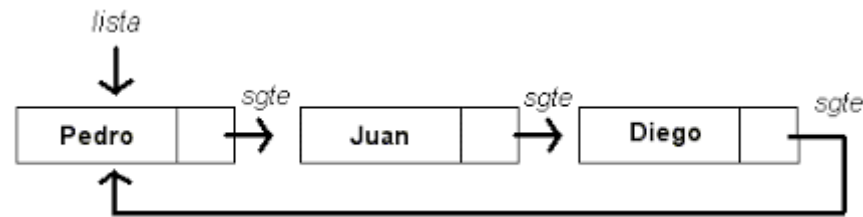
## Lista doblemente enlazada

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.

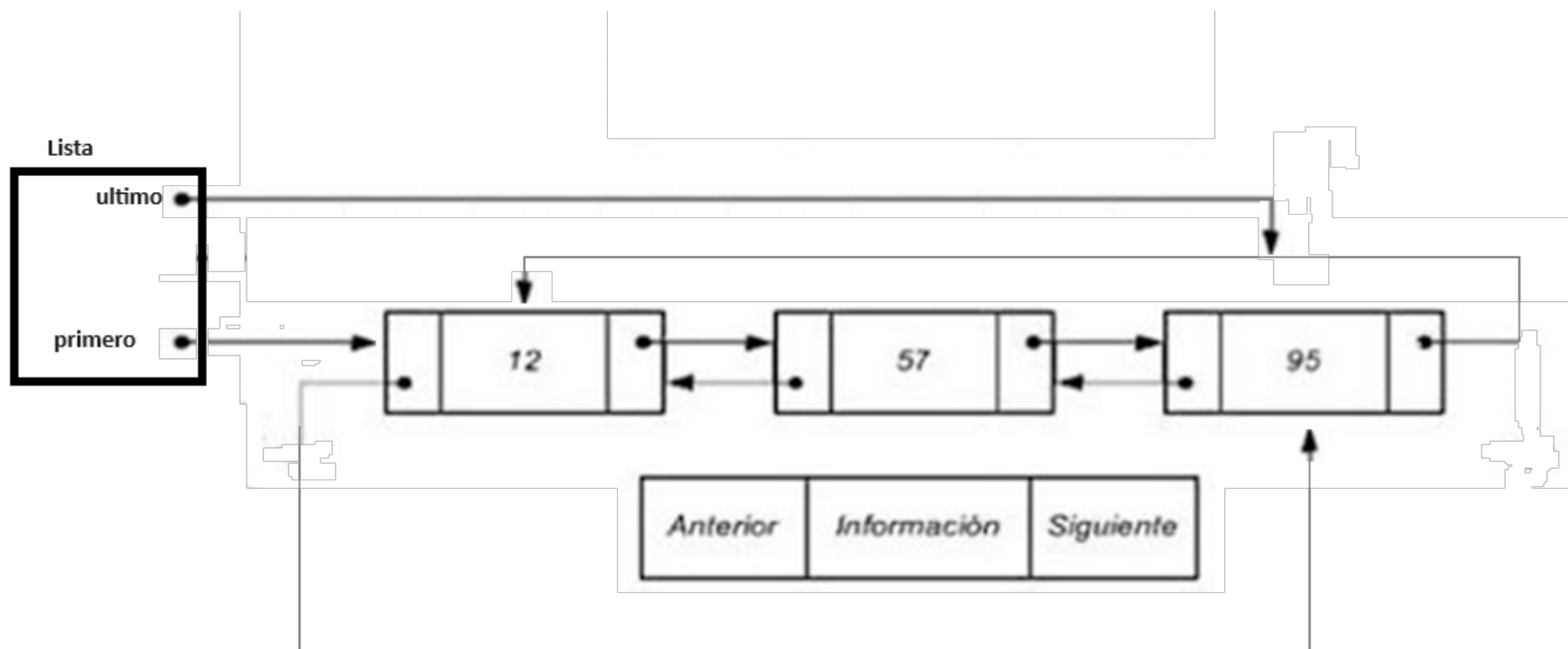


## Lista circular

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



Fin

