



Cátedra Ing. Patricia Calvo

---

# Teoría de Listas

---

Lic. Andrés Juárez

## Índice

1.	<i>Definición</i>	3
2.	<i>Propiedades – Representación</i>	3
3.	<i>Operaciones básicas</i>	4
4.	<i>Distintas formas de implementación</i>	5
4.1	<i>Estructuras estáticas</i>	6
4.2	<i>Estructuras dinámicas</i>	12
4.2.1	<i>Listas Simplemente Enlazadas</i>	12
4.2.2	<i>Listas Doblemente Enlazadas</i>	13
4.3	<i>Otras implementaciones</i>	14
5.	<i>Generalización</i>	15
5.1	<i>Puntero genérico</i>	16
5.2	<i>Plantillas (Templates)</i>	17
6.	<i>Pilas</i>	19
7.	<i>Colas</i>	21
	<i>Apéndices</i>	22
A.	<i>Implementación y ejemplo de uso de listas simplemente enlazadas con dato char.</i>	22
B.	<i>Implementación y ejemplo de uso de listas simplemente enlazadas utilizando templates.</i>	26
C.	<i>Implementación y ejemplo de uso de una Pila dinámica.</i>	31
D.	<i>Implementación y ejemplo de uso de una Cola dinámica.</i>	33
	<i>Nota General</i>	35
8.	<i>Bibliografía</i>	36

## 1. 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.

La estructura lista comprende, a su vez, otras tres estructuras:

- Listas propiamente dichas.
- Pilas.
- Colas.

En el caso de *listas* propiamente dichas, 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.

Hay dos casos especiales de listas que son las *pilas* y las *colas*.

Una *pila* es un caso especial de lista que tiene las siguientes dos restricciones:

- 1) La inserción de un elemento será sólo al final.
- 2) El borrado de un elemento será sólo del final.

Decimos que una pila responde a una estructura de tipo *LIFO* (last in, first out). *Último en entrar primero en salir*.

Una *cola* es un caso especial de lista que tiene las siguientes dos restricciones:

- 1) La inserción de un elemento será sólo al final.
- 2) El borrado de un elemento será sólo del principio.

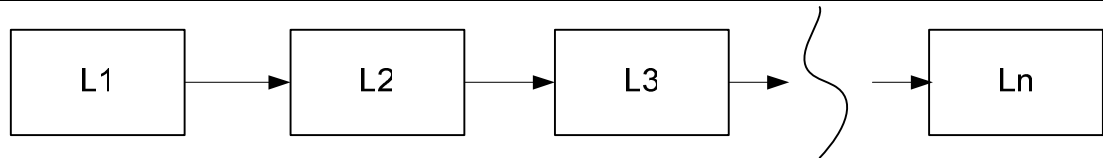
Decimos que una cola responde a una estructura de tipo *FIFO* (first in, first out). *Primero en entrar primero en salir*.

Más adelante veremos la utilidad de estas dos últimas estructuras. Lo que hay que tener en cuenta es que toda pila y toda cola es una lista, pero no sucede al revés. Así como un cuadrado es un rectángulo que tiene todos sus lados iguales, lo que lo convierte en un caso particular de rectángulo; las pilas y colas son casos especiales (con mayores restricciones) de las listas. Cuando hablemos de *listas* nos referiremos a las *propiamente dichas*.

## 2. 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:

$L_1, L_2, L_3, \dots, L_n$



**Nota:** 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.

### 3. 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. insert ( $x$ )
- ii. insert ( $x, p$ )

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

$L1, L2, \dots, Ln$

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

$x, L1, L2, \dots, Ln$

en cambio insert ( $x, n+1$ ) 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. del ( x )
- ii. del ( 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. get\_element ( x.clave)
- ii. get\_element (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.

## **4. 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.

## 4.1 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:

max_length		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.

Una implementación de una lista estática en C++ podría ser la siguiente:

### Archivo lista.h

```
#ifndef LISTA_H_INCLUDED
#define LISTA_H_INCLUDED

// Tipo de dato que contiene la lista
typedef char dato;

// Tamaño máximo de la lista
const int MAX_TAM = 10;

/** Clase Lista_estatica
    Implementada con un vector de elementos
    y un tamaño fijo (MAX_TAM) */
class Lista_estatica
{
private:
    // Vector donde se iran agregando los elementos
    dato lista[MAX_TAM];
    // Tamaño lógico de la lista
    int tope;

public:
```

```

// Constructor
// PRE: ninguna
// POST: crea una lista vacía con tope = 0
Lista_estatica();

// Destructor
// PRE: la lista fue creada
// POST: nada (no tiene código ya que es estática)
~Lista_estatica();

// Inserta un dato
// PRE: lista creada
// POST: Si la lista no está llena
//        - se inserta al final el dato d
//        - tope se incrementa en 1
//        Si la lista está llena no hace nada
void insert(dato d);

// Indica si la lista está vacía o no
// PRE: lista creada
// POST: Devuelve TRUE si la lista está vacía
//        Sino devuelve FALSE
bool lista_vacia();

// Indica si la lista está llena o no
// PRE: lista creada
// POST: Devuelve TRUE si la lista está llena
//        Sino devuelve FALSE
bool lista_llena();

// Devuelve el dato que está en posicion
// PRE: - lista creada
//        - no vacía
//        - 0 < posicion <= tope
// POST: devuelve el dato que está en posición
//        Nota: se toma 1 como la primera posición
dato get_dato(int posicion);

// Borrado de un dato
// PRE: - lista creada
//        - no vacía
//        - 0 < posicion <= tope
// POST: - se borra el dato que está en posición
//        - tope se decrementa en 1
//        Nota: se toma 1 como la primera posición
void del_dato(int posicion);

// Devuelve el tamaño lógico de la lista
// PRE: lista creada
// POST: devuelve el valor de tope
int get_tope();
};
// =====
// Invariantes
// tope >= 0
// tope <= MAX_TAM

#endif // LISTA_H_INCLUDED

```

**Archivo lista.cpp**

```

#include "lista.h"

Lista_estatica::Lista_estatica()
{
    tope = 0;
}

```

```
}

Lista_estatica::~Lista_estatica()
{
}

bool Lista_estatica::lista_llena()
{
    return (tope == MAX_TAM);
}

bool Lista_estatica::lista_vacia()
{
    return (tope == 0);
}

void Lista_estatica::insert(dato d)
{
    if (!(this->lista_llena()))
        lista[tope++] = d;
}

dato Lista_estatica::get_dato(int posicion)
{
    return lista[posicion - 1];
}

void Lista_estatica::del_dato(int posicion)
{
    posicion--;
    tope--;

    while (posicion < tope)
    {
        lista[posicion] = lista[posicion+1];
        posicion++;
    }
}

int Lista_estatica::get_tope()
{
    return tope;
}
```

Y un posible uso de esta lista sería:

#### **Archivo main.cpp**

```
#include <iostream>
#include "lista.h"

using namespace std;

int main()
{
    Lista_estatica l;

    l.insert('A');
    l.insert('H');
    l.insert('E');

    for (int i = 1; i <= l.get_tope(); i++)
    {
```



```
        cout << l.get_dato(i) << endl;
    }

    l.del_dato(2); // Borra la H
    cout << "Listado luego de borrar la H " << endl;

    for (int i = 1; i <= l.get_tope(); i++)
    {
        cout << l.get_dato(i) << endl;
    }

    return 0;
}
```

### Algunas aclaraciones con respecto al ejemplo de implementación

- En primer lugar, el nombre de la clase *Lista\_estatica*, 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 la programación orientada a objetos (POO) 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 *Lista\_estatica* por *Lista\_dinamica*, 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 *get\_dato*) 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).

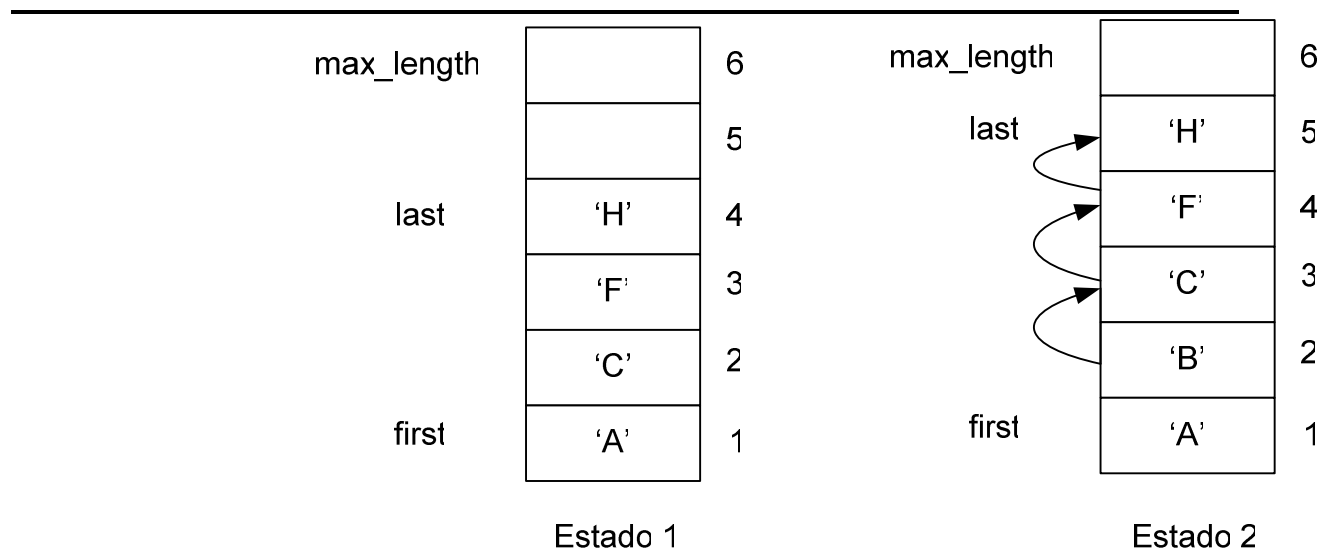
¿Qué problemas tiene una implementación estática?

- 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  $p+1$ ,  $p+2$ , ...,  $tope$ , ubicándolos en las posiciones  $p$ ,  $p+1$ , ...,  $tope-1$ , 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.

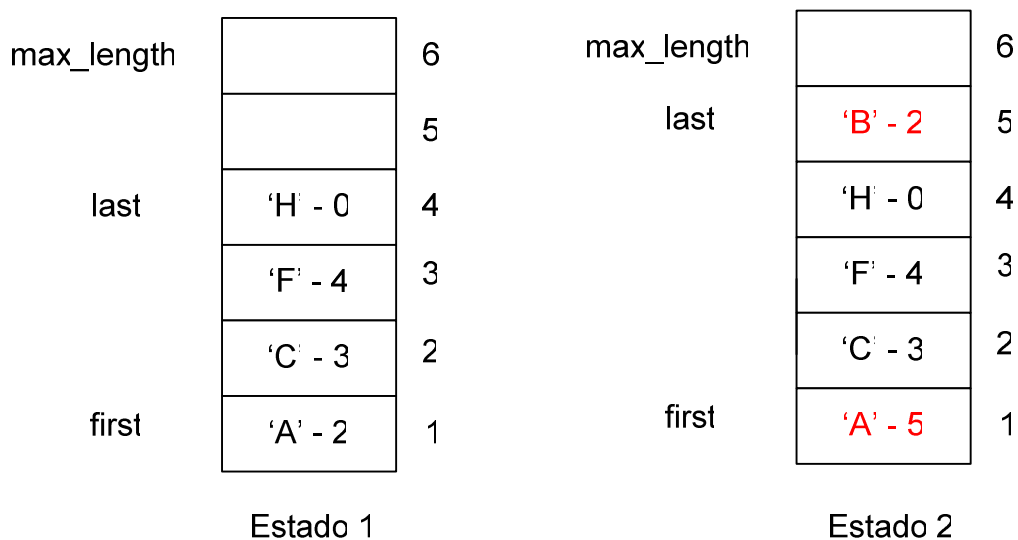
En la siguiente figura se muestran, con flechas, los movimientos que se deberían realizar para pasar del Estado 1 al Estado 2 de una lista ordenada, al insertar la letra ‘B’.



Por supuesto que en ejemplos de dimensión 6, como los de la figura, esto no es ningún problema, pero en los casos reales, en donde se tienen varios cientos o miles (y por qué no millones) de elementos, estas operaciones comienzan a ser costosas.

Hay alternativas para solucionar este último problema manteniendo estructuras estáticas: por ejemplo, cada elemento, además de tener almacenado el dato en sí, tendrá un índice a la posición del siguiente. El último, tendrá un índice que vale 0. Esto simulará una lista dinámica en una implementación que es estática.

La situación del gráfico anterior quedaría, de esta forma:



En el *Estado 1*, el primer elemento está en la posición 1 (no debería por qué serlo), tiene el elemento 'A' e indica que, el siguiente, está en la posición 2 y, así, sucesivamente.

La inserción de 'B' se realiza como en la implementación que realizamos, es decir, un agregado al final. Ahora no tendremos que mover todos los

elementos, solamente debemos modificar el índice que tiene 'A', ya que, ahora, el siguiente es 'B' que está en la posición 5. 'B' "enlaza" con la posición 2, donde está 'C'. El resto no se modifica. En la figura, las celdas que sufren modificaciones, se marcaron en rojo.

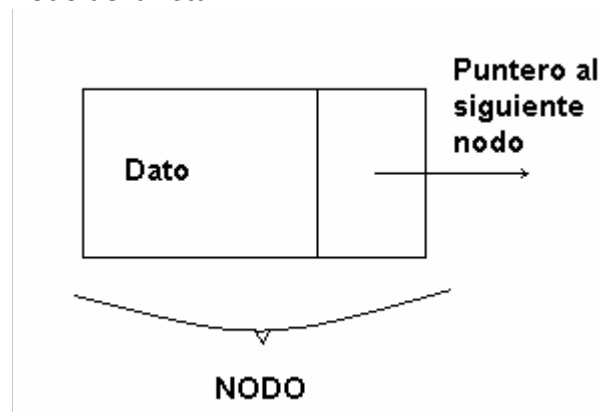
## 4.2 Estructuras dinámicas

### 4.2.1 Listas Simplemente Enlazadas

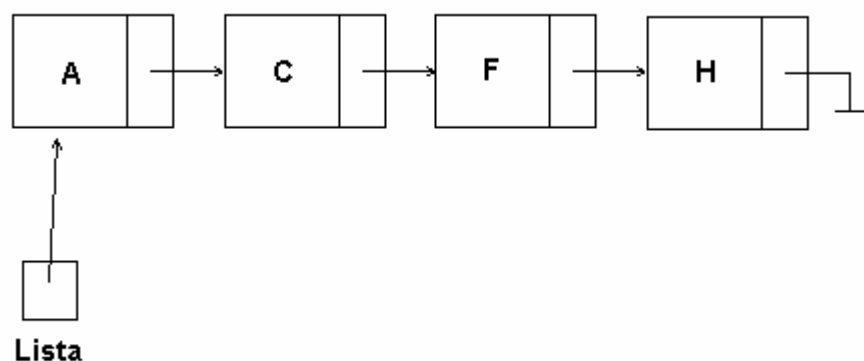
Antes de hablar de listas dinámicas debemos hablar de nodos. Las listas enlazarán nodos. Un nodo es una estructura que tendrá los siguientes datos:

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

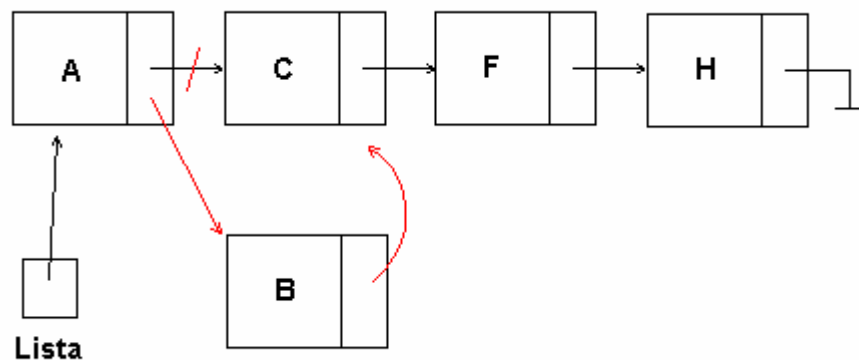
En el caso más simple, el nodo tendrá el elemento y un puntero al siguiente nodo de la lista:



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.

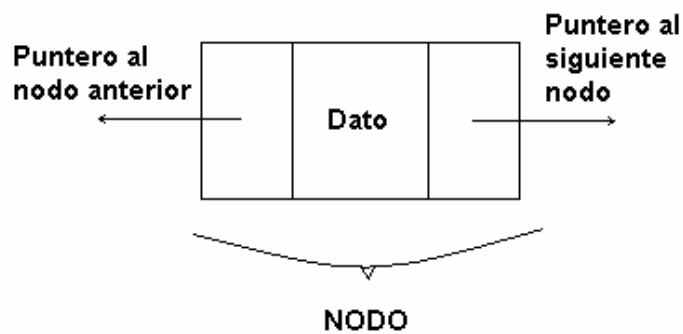
A modo de ejemplo, en el apéndice A se muestra una implementación muy básica (apenas una modificación de la anterior) de esta estructura. Cabe destacar que, a estas alturas, se debe agregar una nueva clase que es *Nodo*. Como se utiliza memoria dinámica cobra especial importancia el destructor, y hay que tener cuidado de liberar la memoria cada vez que se borra un nodo.

#### 4.2.2 Listas Doblemente Enlazadas

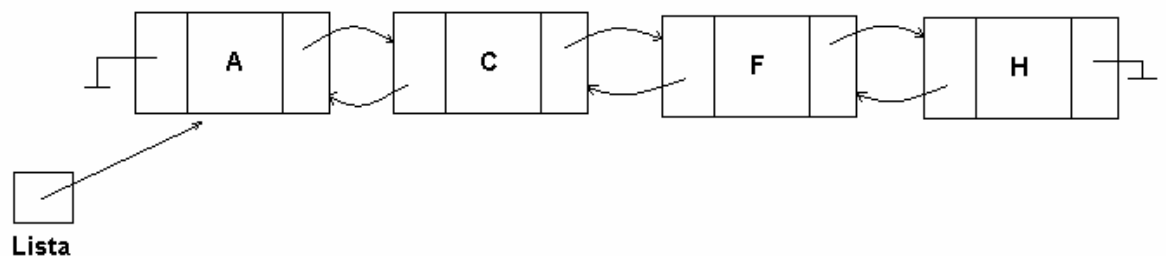
Como seguramente habrán notado, una estructura como la anterior presenta algunas deficiencias. Por ejemplo, si estamos parados en el nodo donde está 'F' y queremos ir al nodo que contiene 'C' (uno anterior), debemos comenzar a recorrer la lista desde el principio, ya que no hay forma de retroceder porque no tenemos ningún puntero que nos lleve al nodo anterior.

Las listas doblemente enlazadas (o bidireccionales) solucionan este punto agregando otro puntero en el nodo, que apunta al nodo anterior. Esta solución tiene un costo: mayor consumo de memoria y mayor complejidad (en general) a la hora de programar.

Gráficamente, un nodo se puede representar como se muestra a continuación:



Y la lista:



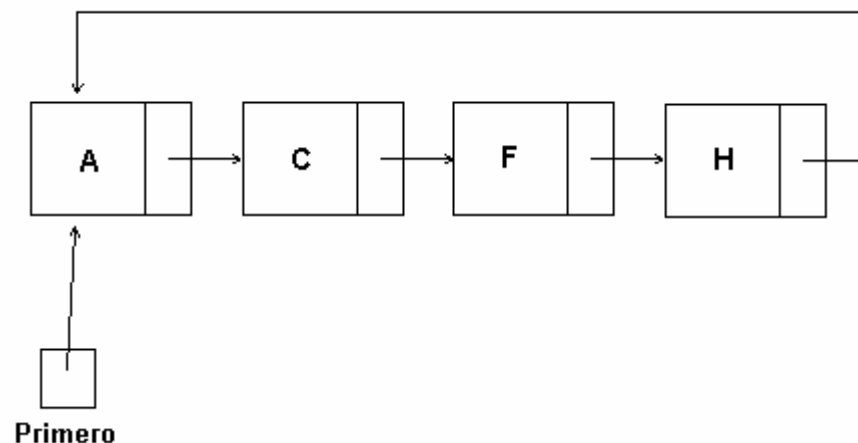
Hay que tener en cuenta que, al insertar un nuevo nodo, se deben ajustar cuatro punteros: los dos del nuevo nodo, el anterior del siguiente y el posterior del anterior.

Una implementación de este tipo tendría sentido en aplicaciones donde se debe estar accediendo frecuentemente tanto a elementos anteriores de un nodo dado como posteriores, como pueden ser los algoritmos de compresión o suavizado de curvas, etc.

También tendría sentido agregar un atributo más a la clase, otro puntero, que sería el *actual*. Un puntero que, en cada momento, esté señalando el nodo donde se está poniendo el foco.

### 4.3 Otras implementaciones

Una tipo de implementación importante de mencionar son las *listas circulares*. Una *lista circular* es una lista en donde el último nodo apunta al primero, como se aprecia en el siguiente gráfico.



### ¿Cuál es la ventaja / utilidad de una implementación así?

Este tipo de implementaciones son muy útiles para el tratamiento de datos que se reciben en forma continua, como puede ser, por ejemplo, datos que se reciben constantemente de un sensor, el cual puede tomar temperaturas, flujos, etc. Como son procesos continuos en el tiempo, una lista circular de  $n$  nodos nos permite tener siempre vigentes los últimos  $n$  datos.

## 5. Generalización

Hasta acá vimos un pantallazo de las principales implementaciones de listas, pero no nos detuvimos en algo muy importante: la generalización o programación genérica.

Los ejemplos de implementaciones que se dieron utilizaban, como dato, un carácter, por eso definimos

```
typedef char Dato;
```

y, luego, se utilizó *Dato* como tipo en donde se necesitara, ya sea como parámetro o como algo que devuelve cierto método.

Si necesitáramos utilizar una lista de enteros, sólo deberíamos cambiar la palabra *char* por *int* y no tendríamos ningún problema.

Sin embargo, se puede dar que necesitemos una lista de *char* y otra de *int*. En este punto se comienzan a complicar las cosas, ya que necesitaríamos dos tipos de listas distintas. También, podríamos necesitar una lista de elementos más complejos, como los datos de un empleado (legajo, nombre, dirección, etc.). Como habíamos comentado, no sería bueno, en ningún método, devolver un dato tan grande y complejo.

Por otro lado, ¿si necesitamos nuevas primitivas que nos devuelvan algún valor especial cuando buscamos un elemento en una lista y no lo encontramos?

La solución a esta última pregunta, como se había anticipado, es devolver una referencia o puntero al dato en lugar del propio dato, de esta forma, cuando no se encuentra un elemento se puede devolver un valor nulo. Pero los otros problemas no los podemos resolver tan fácil.

## 5.1 Puntero genérico

C++ (como varios otros lenguajes) nos permite utilizar un puntero genérico (o puntero a nada). Que es de tipo *void*. *Void* significa *vacío*, por lo que se puede interpretar que un puntero a *void* carece de tipo, no apunta o no está atado a ningún tipo en especial, por lo que puede apuntar a cualquier tipo. De esta forma, estos punteros serán genéricos, por lo que abarcarían (servirían para) cualquier tipo dato.

El problema con este tipo de implementación es que debemos estar casteando (conversión de tipos) las devoluciones de los métodos.

La implementación prácticamente no cambia en nada, salvo por el tipo de dato que ahora será un puntero a void:

```
typedef void* Dato;
```

Pero, hay que tener cuidado en el uso porque debemos recordar dos cosas:

- Tanto los datos que se pasan en forma de parámetros como los devueltos son, ahora, punteros.
- Los punteros a los datos que nos devuelvan los métodos se deben castear para poder utilizarlos.

El siguiente código es un ejemplo de uso de este tipo de listas, se verá que se pueden utilizar para distintos tipos de datos (en este caso enteros y strings).

### Archivo main.cpp

```
#include "lista.h"
#include <string>

using namespace std;

int main()
{
    cout << "==== Lista de enteros ==== << endl;

    int a = 7, b = 8, c = 3;
    Lista_void li;

    // No se pasan los datos sino sus direcciones
    li.insert(&a);
    li.insert(&b);
    li.insert(&c);

    for (unsigned i = 1; i <= li.get_tam(); i++)
        // El primer asterisco es para acceder al dato
        // (recordar que devuelve un puntero al dato)
        // El "(int*)" es el casteo porque el
        // puntero que devuelve es genérico
        cout << *(int*)li.get_dato(i) << endl;

    // Ahora, la misma lista se puede utilizar con otros
    // tipos de datos, como strings
    cout << "==== Lista de strings ==== << endl;

    string s1 = "siete", s2 = "ocho", s3 = "tres";
```



```
Lista_void ls;  
  
ls.insert(&s1);  
ls.insert(&s2);  
ls.insert(&s3);  
  
for (unsigned i = 1; i <= ls.get_tam(); i++)  
    cout << *(string*)ls.get_dato(i) << endl;  
  
return 0;  
}
```

Nada nos impediría escribir y ejecutar la siguiente porción de código:

```
int a = 7, b = 8, c = 3;  
string s1 = "hola";  
Lista_void l;  
  
l.insert(&a);  
l.insert(&s1);  
l.insert(&b);  
l.insert(&c);
```

De esta forma estaríamos poniendo en la misma lista punteros a enteros y un puntero a un string. Si bien esto sería aceptado y no arrojaría ningún error, debemos recordar que, cuando deseemos recuperar el elemento que está en el tercer nodo (en este ejemplo, recordar que se inserta al principio), debemos castearlo como (string\*) y no como (int\*) como al resto de los elementos.

Obviamente, uno no puede estar recordando o llevando nota sobre qué tipos de elementos están en cada uno de los nodos, por lo que esta solución no es recomendable.

Sin embargo es importante tener en cuenta lo siguiente: la porción de código escrita anteriormente podría deberse a un error en la programación. Es decir, el programador podría haber deseado insertar el puntero de *s1* en otra lista, una de strings y no en la misma lista donde está almacenando los enteros. Sin embargo el compilador no da ninguna advertencia sobre diferencias de tipos, ¡porque no la hay! Los nodos de la lista están preparados para aceptar direcciones de cualquier clase de elementos, como le pasamos una dirección lo toma como algo correcto, sin importarle si esa dirección es a un string, a un entero o a cualquier otro objeto.

Para salvar este problema, es decir, para poder tener una verificación de tipos, están las plantillas (templates).

## 5.2 Plantillas (Templates)

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*.

Se debe agregar, antes de definir la clase

```
template < typename dato >    // Se agrega
class Lista_estatica
{
    ...
};
```

Luego, cuando vayamos a utilizar la lista, debemos indicarle de qué tipo la queremos. Por ejemplo:

```
Lista_estatica < char >    lc;        // Lista de chars
Lista_estatica < string >  ls;        // Lista de strings
Lista_estatica < string >  ls2;       // Otra lista de strings
```

Cuando se compila, se resuelve el problema del tipo indefinido en un proceso que se llama especialización. ¿Qué es lo que hace? En el ejemplo que acabamos de dar, observa que necesita una lista para un tipo *char*, por lo que genera, automáticamente, el código por nosotros. Es decir, reemplaza *dato* por *char* en todas sus ocurrencias, generando el código completo.

Luego, cuando compile la segunda línea, hace lo mismo pero con strings, es decir, genera nuevamente todo el código pero reemplazando *string* en donde figure *dato*.

Por supuesto, en la tercera línea, en la declaración de *ls2*, no vuelve a generar código porque ya tiene código de listas para strings.

¿Cuáles son las ventajas de los templates?

- Generan código en forma automática por nosotros.
- Hay verificación de tipos. Ya no podría, en una lista de enteros, agregar un string como en el ejemplo de la sección 5.1. Esto, lejos de ser una molestia, es una ventaja, porque es una seguridad que el compilador verifique que los tipos sean correctos.
- Como la definición de tipos se resuelve en tiempo de compilación, los ejecutables son más rápidos que los generados utilizando herencia y polimorfismo.

#### Algunas características a tener en cuenta

- Un template no existe (no genera código) hasta que se instancia. Es decir, hasta que no se crea algún objeto indicando el tipo no genera ningún código.
- Una consecuencia del punto anterior es que se debe generar todo el código en el archivo punto h. Por lo tanto no se separarán las declaraciones por un lado (archivo punto h) y las definiciones por otro (archivo punto cpp). De todas formas, es recomendable seguir separando las declaraciones de las definiciones, aunque lo incluyamos todo en el mismo archivo. De lo contrario se generarían todos métodos *inline*, que tienen la ventaja de ser más rápidos en la ejecución pero generan un código mucho mayor, ya que se copian cada vez que se invocan.
- Otra característica del uso de templates es que se necesita indicar el parámetro cada vez que se indique la clase cuando se definen los métodos (esto se ve más claramente en el código que figura en el apéndice B).
- Finalmente, cabe destacar que es indistinto utilizar la palabra *typename* como *class*. Pero se prefiere la primera ya que el parámetro no tiene por qué ser una clase (objeto) sino cualquier tipo.

## 6. Pilas

Una pila, como habíamos anticipado, es un caso particular de lista, en donde el ingreso de datos se realiza sólo por el final o cima de la pila, y la extracción, también. De esta manera responden a una estructura de tipo LIFO (el último en entrar o llegar es el primero en salir).

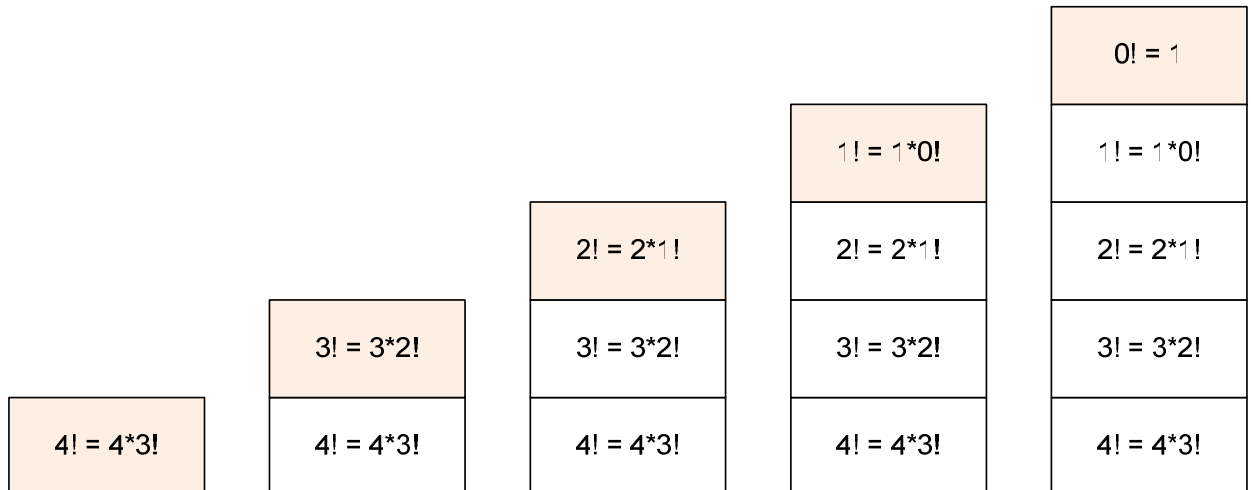
¿Con qué finalidad podríamos necesitar una estructura con estas características?

Hay determinados algoritmos que se modelan correctamente con una pila, por ejemplo, los que utilizan recursividad. Veamos cómo funcionaría un algoritmo recursivo que calcule el factorial de 4.

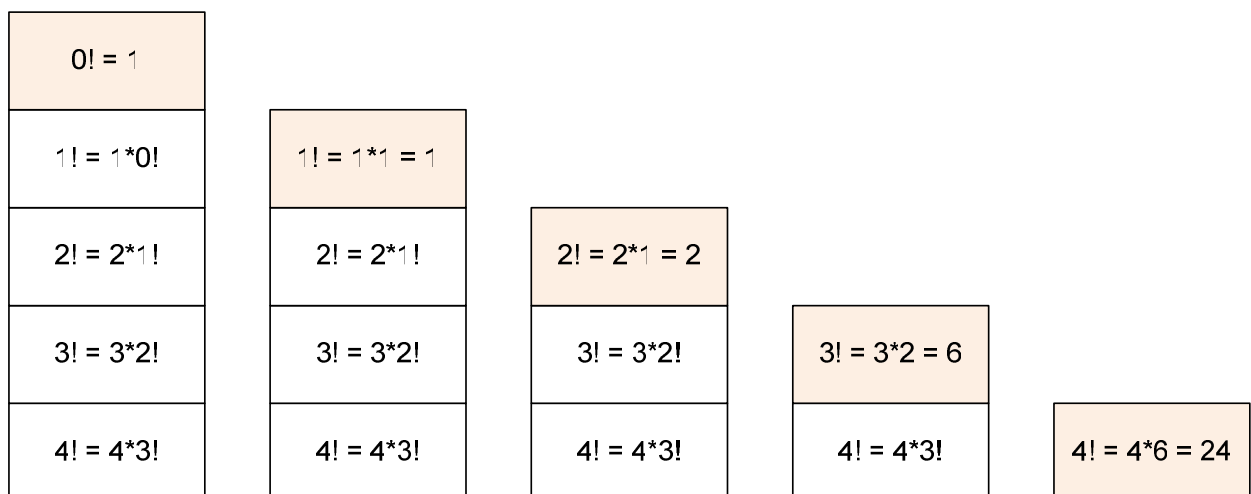
Para su cálculo, el factorial de cuatro lo podemos expresar como cuatro por el factorial de su anterior, tres. De esta forma, tres ingresa en la pila recursiva, en donde se aplica el

mismo procedimiento. Este algoritmo continúa hasta llegar a cero, que sería el único número del cual conocemos su factorial. Cuando se comienza a resolver las incógnitas, se deben ir resolviendo de atrás para adelante.

En la siguiente figura vemos cómo se va generando la pila.



Y, luego, cómo se va desapilando:



Hay que notar que los elementos que están debajo de la cima están a la espera de los resultados de los elementos superiores. Es decir, el elemento que está en la posición 1 necesita que se resuelva el de la posición 2 y, así, sucesivamente.

Las diferentes implementaciones que se pueden aplicar a una estructura de tipo *Pila* son las mismas que las que vimos con listas: estáticas, dinámicas, con templates, etc. Desde ya que una implementación del tipo *doblemente enlazada* no tendría sentido porque no se necesita ir para “atrás” y “adelante”. Un ejemplo de implementación y uso se muestra en el apéndice C.

## 7. Colas

Al igual que las *Pilas*, las *Colas* son casos especiales de listas, donde responden a una estructura del tipo FIFO (último en entrar o llegar, es el último en salir). Modelan muchas situaciones de la vida diaria, y tienen el mismo sentido de la palabra que en el uso habitual. Cuando uno necesita hacer algún trámite, por ejemplo, llega al lugar, se coloca al final de la cola que hubiere y debe esperar su turno mientras se atienden a las personas que están delante. Un ejemplo de aplicación típico de la informática es el de varias máquinas conectadas a una impresora, en donde los distintos usuarios envían a imprimir archivos desde sus terminales. La solicitud ingresa a una cola en donde se van procesando de una en una cada solicitud, de lo contrario, el resultado sería una mixtura de los distintos archivos que serían ininteligibles.

Por supuesto, en su forma más sencilla una cola respeta siempre el orden de llegada, aunque hay ciertas variantes en donde se pueden determinar distintas prioridades que alteren dicho orden. Por ejemplo, una solicitud de impresión tendrá prioridad más alta que el proceso de escaneo de toda la máquina en busca de virus.

La implementación es muy similar a la de una pila. Un ejemplo se puede observar en el apéndice D.

## Apéndices

### A. Implementación y ejemplo de uso de listas simplemente enlazadas con dato char.

#### Archivo nodo.h

```
#ifndef NODO_H_INCLUDED
#define NODO_H_INCLUDED

#include <iostream>
// Tipo de dato char
typedef char Dato;

class Nodo
{
private:
    Dato dato;        // Dato a almacenar
    Nodo* psig;       // Puntero a otro nodo

public:
    // Constructor con parametro
    // PRE: Ninguna
    // POST: Crea un nodo con el dato d
    //       y el puntero a NULL
    Nodo(Dato d);

    // Destructor
    // PRE:  Nodo creado
    // POST: No hace nada
    ~Nodo();

    // Setea el dato (lo cambia)
    // PRE:  el nodo tiene que estar creado
    // POST: El nodo queda con el dato d
    void set_dato(Dato d);

    // Obtener el dato
    // PRE:  nodo creado
    // POST: devuelve el dato que contiene el nodo
    Dato get_dato();

    // Setear el puntero al siguiente nodo
    // PRE:  nodo creado
    // POST: el puntero al siguiente nodo apuntará a ps
    void set_siguiente(Nodo* ps);

    // Obtener el puntero al nodo siguiente
    // PRE:  nodo creado
    // POST: Devuelve el puntero al siguiente nodo
    //       Si es el último devuelve NULL
    Nodo* get_siguiente();

};

#endif // NODO_H_INCLUDED
```

**Archivo nodo.cpp**

```
#include "nodo.h"

Nodo::Nodo(Dato d)
{
    std::cout << "Se construye el nodo " << (unsigned)this << std::endl;
    dato = d;
    psig = 0;
}

Nodo::~Nodo()
{
    std::cout << "Se destruye el nodo " << (unsigned)this << std::endl;
}

void Nodo::set_dato(Dato d)
{
    dato = d;
}

Dato Nodo::get_dato()
{
    return dato;
}

void Nodo::set_siguiete(Nodo* ps)
{
    psig = ps;
}

Nodo* Nodo::get_siguiete()
{
    return psig;
}
```

**Nota:** se utilizan carteles para indicar cuándo se construye un nodo y cuándo se destruye y sus direcciones. Obviamente, estos carteles deben ser borrados pero es una buena práctica utilizarlos en las pruebas, con el fin de controlar la memoria pedida y liberada.

**Archivo lista.h**

```
#ifndef LISTA_H_INCLUDED
#define LISTA_H_INCLUDED

#include "nodo.h"

class Lista_SE
{
private:
    // Primer elemento de la lista
    Nodo* primero;
    // Tamaño de la lista
    unsigned tam;

public:
    // Constructor
    // PRE: ninguna
    // POST: construye una lista vacia
    //      - primero apunta a nulo
    //      - tam = 0
    Lista_SE();
}
```

```

// Destructor
// PRE:  lista creada
// POST: Libera todos los recursos de la lista
~Lista_SE();

// La lista es vacía?
// PRE:  lista creada
// POST: devuelve verdadero si la lista es vacia
//       falso de lo contrario
bool lista_vacia();

// Agregar un elemento a la lista
// PRE:  lista creada
// POST: agrega un dato (dentro de un nodo) al final
//       incrementa tam en 1
void insert(Dato d);

// Obtener el dato que está en la posición pos
// PRE:  - lista creada y no vacia
//       - 0 < pos <= tam
// POST: devuelve el dato que esta en la posicion pos
//       se toma 1 como el primero
Dato get_dato(unsigned pos);

// Borrado del nodo que está en la posición pos
// PRE:  - lista creada y no vacia
//       - 0 < pos <= tam
// POST: libera el nodo que esta en la posición pos
//       se toma 1 como el primero
void del_dato(unsigned pos);

// Obtener el tamaño de la lista
// PRE:  Lista creada
// POST: Devuelve tam (cantidad de nodos de la lista)
unsigned get_tam();
};

#endif // LISTA_H_INCLUDED

```

### Archivo lista.cpp

```

#include "lista.h"

Lista_SE::Lista_SE()
{
    primero = 0;
    tam      = 0;
}

Lista_SE::~~Lista_SE()
{
    while (!(this->lista_vacia()))
        this->del_dato(1);
}

bool Lista_SE::lista_vacia()
{
    return (primero == 0);
}

void Lista_SE::insert(Dato d)
{
    Nodo* pnode = new Nodo(d);
    Nodo* paux  = primero;

    if (this->lista_vacia())
        primero = pnode;
}

```



```
        else
        {
            while (paux->get_siguiete())
                paux = paux->get_siguiete();

            paux->set_siguiete(pnodo);
        }
        tam++;
    }

Dato Lista_SE::get_dato(unsigned pos)
{
    Nodo* paux = primero;
    unsigned i = 1;

    while (i < pos && paux->get_siguiete())
    {
        paux = paux->get_siguiete();
        i++;
    }

    return paux->get_dato();
}

void Lista_SE::del_dato(unsigned pos)
{
    Nodo* paux = primero;

    if (pos == 1 || !(primero->get_siguiete()))
    {
        primero = paux->get_siguiete();
    }
    else
    {
        unsigned i = 1;
        Nodo* pant;

        while (i < pos && paux->get_siguiete())
        {
            pant = paux;
            paux = paux->get_siguiete();
            i++;
        }

        pant->set_siguiete(paux->get_siguiete());
    }

    delete paux;
    tam--;
}

unsigned Lista_SE::get_tam()
{
    return tam;
}
```

Un posible uso de esta lista sería:

**Archivo main.cpp**

```
#include "lista.h"

using namespace std;

int main()
{
    Lista_SE l;

    l.insert('A');
    l.insert('C');
    l.insert('F');
    l.insert('H');

    for (unsigned i = 1; i <= l.get_tam(); i++ )
    {
        cout << l.get_dato(i) << endl;
    }

    // Se borra el segundo nodo
    l.del_dato(2);
    for (unsigned i = 1; i <= l.get_tam(); i++ )
    {
        cout << l.get_dato(i) << endl;
    }

    return 0;
}
```

**B. Implementación y ejemplo de uso de listas simplemente enlazadas utilizando templates.****Archivo nodo.h**

```
#ifndef NODO_H_INCLUDED
#define NODO_H_INCLUDED

template < typename Dato >
class Nodo
{
private:
    Dato dato;        // Dato a almacenar
    Nodo* psig;       // Puntero a otro nodo

public:
    // Constructor con parametro
    // PRE: Ninguna
    // POST: Crea un nodo con el dato d
    //        y el puntero a NULL
    Nodo(Dato d);

    // Destructor
    // PRE:  Nodo creado
    // POST: No hace nada
    ~Nodo();

    // Setea el dato (lo cambia)
    // PRE:  el nodo tiene que estar creado
    //        d tiene que ser un dato válido
```

```

        // POST: el nodo queda con el dato d
        void set_dato(Dato d);

        // Setear el puntero al siguiente nodo
        // PRE:  nodo creado y ps válido
        // POST: el puntero al siguiente apuntará a ps
        void set_sig(Nodo* ps);

        // Obtener el dato
        // PRE:  nodo creado
        // POST: devuelve el dato que contiene el nodo
        Dato get_dato();

        // Obtener el puntero al nodo siguiente
        // PRE:  nodo creado
        // POST: Devuelve el puntero al siguiente nodo
        //        si es el último devuelve NULL
        Nodo* get_sig();

        // ¿Hay un siguiente?
        // PRE:  nodo creado
        // POST: V si tiene sig. F sino
        bool tiene_sig();
};

// Constructor con parametro
template < typename Dato >
Nodo<Dato>::Nodo(Dato d)
{
    dato = d;
    psig = 0;
}

// Destructor
template < typename Dato >
Nodo<Dato>::~~Nodo()
{
    // No hace nada
}

// Setear el dato
template < typename Dato >
void Nodo<Dato>::set_dato(Dato d)
{
    dato = d;
}

// Setear el ptr al sig
template < typename Dato >
void Nodo<Dato>::set_sig(Nodo* ps)
{
    psig = ps;
}

// Devolver el dato
template < typename Dato >
Dato Nodo<Dato>::get_dato()
{
    return dato;
}

// Devolver el siguiente
template < typename Dato >
Nodo<Dato>* Nodo<Dato>::get_sig()
{
    return psig;
}

```

```

}

// Tiene siguiente?
template < typename Dato >
bool Nodo<Dato>::tiene_sig()
{
    return (psig != 0);
}

#endif // NODO_H_INCLUDED

```

### Archivo lista.h

```

#ifndef LISTA_H_INCLUDED
#define LISTA_H_INCLUDED

#include <iostream>
#include "nodo.h"

template < typename Dato >
class Lista
{
private:
    // Primer elemento de la lista
    Nodo<Dato>* primero;
    // Tamaño de la lista
    unsigned tam;

public:
    // Constructor
    // PRE: Ninguna
    // POST: construye una lista vacía
    //      - primero apunta a nulo
    //      - tam = 0
    Lista();

    // Destructor
    // PRE: Lista creada
    // POST: Libera todos los recursos de la lista
    ~Lista();

    // Agregar un elemento a la lista
    // PRE: lista creada y d válido
    // POST: agrega un dato dentro de un nodo al principio
    //      - modifica el primero
    //      - tam se incrementa en 1
    void insert(Dato d);

    // Obtener el tamaño de la lista
    // PRE: Lista creada
    // POST: devuelve el tamaño de la lista (cantidad de nodos)
    unsigned get_tam();

    // Obtener el dato que está en la posición pos
    // PRE: - lista creada y no vacía
    //      - 0 < pos <= tam
    // POST: devuelve el dato que está en la posición pos
    //      se toma 1 como el primero
    Dato get_dato(unsigned pos);

    // ¿Lista vacía?
    // PRE: Lista creada
    // POST: T si es vacía, F sino
    bool lista_vacia();

    // Borrado del nodo que está en la posición pos

```

```

        // PRE:  - lista creada y no vacía
        //        - 0 < pos <= tam
        // POST: libera el nodo que está en la posición pos
        //        se toma 1 como el primero
        void del_dato(unsigned pos);

};

// Constructor
template < typename Dato >
Lista<Dato>::Lista()
{
    primero = 0;
    tam      = 0;
}

// Destructor
template < typename Dato >
Lista<Dato>::~~Lista()
{
    while (!this->lista_vacia())
    {
        this->del_dato(1);
    }
}

// Devuelve T si la lista esta vacia
// Sino F
template < typename Dato >
bool Lista<Dato>::lista_vacia()
{
    return (tam == 0);
}

// Colocar, al principio, un nuevo nodo con el dato d
template < typename Dato >
void Lista<Dato>::insert(Dato d)
{
    Nodo<Dato>* nuevo = new Nodo<Dato>(d);

    if (!(this->lista_vacia()))
    {
        nuevo->set_sig(primeros);
    }

    primero = nuevo;
    tam++;
}

// Devuelve el dato que esta en la posición pos
template < typename Dato >
Dato Lista<Dato>::get_dato(unsigned pos)
{
    Nodo<Dato>* aux = primero;

    if (!lista_vacia())
    {
        for (unsigned cont = 1; (cont < pos && aux); cont++)
            aux = aux->get_sig();

        return aux->get_dato();
    }

    return 0;
}

// Devuelve la cantidad de nodos de la Lista

```

```

template < typename Dato >
unsigned Lista<Dato>::get_tam()
{
    return tam;
}

// Elimina el dato que esta en la posición pos
template < typename Dato >
void Lista<Dato>::del_dato(unsigned pos)
{
    Nodo<Dato>* aux = primero;

    if (!lista_vacia())
    {
        if (pos == 1)
        {
            primero = primero->get_sig();
        }
        else
        {
            for (unsigned cont = 0; (cont < pos-1 && aux->get_sig()); cont++)
                aux = aux->get_sig();

            aux->set_sig(aux->get_sig());
        }

        Nodo<Dato>* borrar = aux;
        std::cout << "Se elimina el nodo con el dato " << borrar->get_dato() <<
std::endl;
        delete borrar;
        tam--;
    }
}

#endif // LISTA_H_INCLUDED

```

### Archivo main.cpp

```

#include "lista.h"
#include <string>

using namespace std;

int main()
{
    cout << "==== Lista de enteros =====" << endl;
    Lista<int> l;

    l.insert(7);
    l.insert(8);
    l.insert(3);

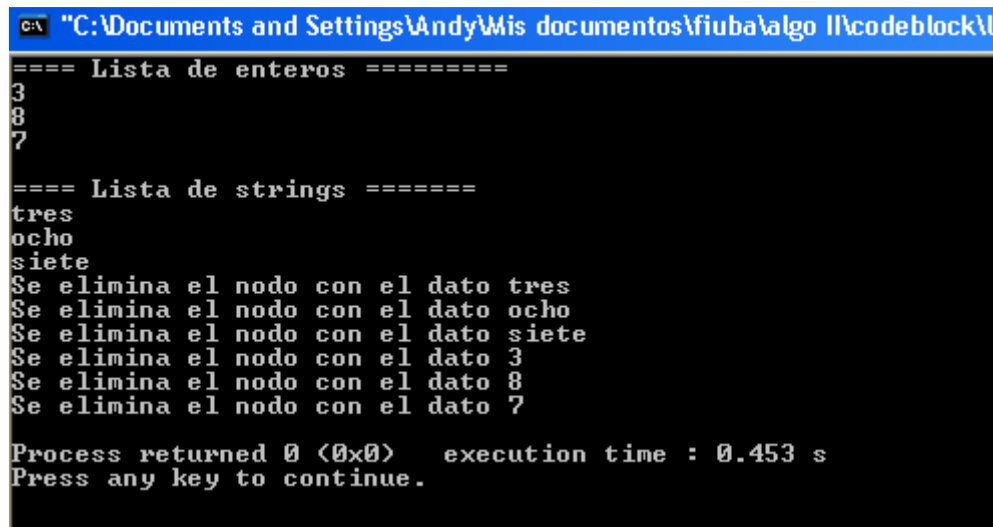
    for (unsigned i=1; i <= l.get_tam(); i++)
        cout << l.get_dato(i) << endl;

    cout << endl << "==== Lista de strings =====" << endl;
    Lista<string> ls;
    ls.insert("siete");
    ls.insert("ocho");
    ls.insert("tres");

    for (unsigned i=1; i <= ls.get_tam(); i++)
        cout << ls.get_dato(i) << endl;

    return 0;
}

```

**Salida por pantalla**

```
C:\Documents and Settings\Andy\Mis documentos\fiuba\algo II\codeblock\
==== Lista de enteros =====
3
8
7

==== Lista de strings =====
tres
ocho
siete
Se elimina el nodo con el dato tres
Se elimina el nodo con el dato ocho
Se elimina el nodo con el dato siete
Se elimina el nodo con el dato 3
Se elimina el nodo con el dato 8
Se elimina el nodo con el dato 7

Process returned 0 (0x0)   execution time : 0.453 s
Press any key to continue.
```

**C. Implementación y ejemplo de uso de una Pila dinámica.****Archivo pila.h**

```
#ifndef PILA_H_INCLUDED
#define PILA_H_INCLUDED

#include "nodo.h"

class Pila
{
private:
    // Primer elemento de la pila
    Nodo* primero;

public:
    // Constructor
    // PRE: ninguna
    // POST: construye una pila vacía
    //       - primero apunta a nulo
    Pila();

    // Destructor
    // PRE: pila creada
    // POST: Libera todos los recursos de la pila
    ~Pila();

    // ¿La pila es vacía?
    // PRE: pila creada
    // POST: devuelve verdadero si la pila es vacía
    //       falso de lo contrario
    bool pila_vacia();

    // Agregar un elemento a la pila
    // PRE: pila creada
    // POST: agrega un dato (dentro de un nodo) al principio
    void insert(Dato d);
};
```

```

        // Obtener el dato que está en la cima
        // PRE:  - pila creada y no vacía
        // POST: devuelve el dato que está en la cima
        Dato get_dato();

        // Borrado del nodo que está en la cima
        // PRE:  - pila creada y no vacía
        // POST: libera el nodo que está en la cima
        void del_dato();

};

#endif // PILA_H_INCLUDED

```

**Archivo pila.cpp**

```

#include "pila.h"

Pila::Pila()
{
    primero = 0;
}

Pila::~~Pila()
{
    while (!(this->pila_vacia()))
        this->del_dato();
}

bool Pila::pila_vacia()
{
    return (primero == 0);
}

void Pila::insert(Dato d)
{
    Nodo* pnode = new Nodo(d);
    pnode->set_siguiete(primero);
    primero = pnode;
}

Dato Pila::get_dato()
{
    return primero->get_dato();
}

void Pila::del_dato()
{
    Nodo* paux = primero;

    primero = paux->get_siguiete();
    delete paux;
}

```



**Archivo main.cpp**

```
#include "pila.h"

using namespace std;

int main()
{
    Pila p;

    p.insert('A');
    p.insert('H');
    p.insert('B');

    while (!p.pila_vacia())
    {
        cout << p.get_dato() << endl;
        p.del_dato();
    }

    return 0;
}
```

**Notas**

- Los archivos nodo.h y nodo.cpp no se incluyeron porque son los mismos que los utilizados en la implementación de *listas simplemente enlazadas*.
- Se utilizó un char como dato para concentrarnos en la implementación y no estar pendiente de otras cosas.
- El puntero *primero* siempre apunta a la cima de la pila.
- Como se puede observar del código, la implementación de una pila es mucho más sencilla que la de una lista.

**D. Implementación y ejemplo de uso de una Cola dinámica.****Archivo cola.h**

```
#ifndef COLA_H_INCLUDED
#define COLA_H_INCLUDED

#include "nodo.h"

class Cola
{
private:
    // Primer elemento de la cola
    Nodo* primero;
    // Ultimo elemento de la cola
    Nodo* ultimo;

public:
    // Constructor
    // PRE: ninguna
    // POST: construye una cola vacía
    // - primero y ultimo apuntan a nulo
    Cola();

    // Destructor
```

```

// PRE: cola creada
// POST: Libera todos los recursos de la cola
~Cola();

// ¿La cola es vacía?
// PRE: cola creada
// POST: devuelve verdadero si la cola es vacía
//        falso de lo contrario
bool cola_vacia();

// Agregar un elemento a la cola
// PRE: cola creada
// POST: agrega un dato (dentro de un nodo) al final
void insert(Dato d);

// Obtener el dato que está al principio
// PRE: - cola creada y no vacía
// POST: devuelve el dato que está al principio
Dato get_dato();

// Borrado del nodo que está al principio
// PRE: - cola creada y no vacía
// POST: libera el nodo que está al principio
void del_dato();

};

#endif // COLA_H_INCLUDED

```

### Archivo cola.cpp

```

#include "cola.h"

Cola::Cola()
{
    primero = ultimo = 0;
}

Cola::~~Cola()
{
    while (!(this->cola_vacia()))
        this->del_dato();
}

bool Cola::cola_vacia()
{
    return (primero == 0);
}

void Cola::insert(Dato d)
{
    Nodo* pnodo = new Nodo(d);

    if (this->cola_vacia())
        primero = pnodo;
    else
        ultimo->set_siguiente(pnodo);

    ultimo = pnodo;
}

Dato Cola::get_dato()

```

```
{
    return primero->get_dato();
}

void Cola::del_dato()
{
    Nodo* paux = primero;

    primero = paux->get_siguiete();
    delete paux;
}
```

### Archivo main.cpp

```
#include "cola.h"

using namespace std;

int main()
{
    Cola c;

    c.insert('A');
    c.insert('H');
    c.insert('B');

    while (!c.cola_vacia())
    {
        cout << c.get_dato() << endl;
        c.del_dato();
    }

    return 0;
}
```

### Notas

- Al igual que en la *Pila* los archivos nodo.h y nodo.cpp son los mismos que el de *listas simplemente enlazadas*.
- También se utilizó *char* como dato con el fin de centrarnos en la implementación de la *cola* y no en otras cuestiones.
- Se utiliza un puntero al último nodo para lograr mayor eficiencia (más rapidez a la hora de insertar un nodo), pero no es necesario.

### Nota General

Las implementaciones mostradas son sólo orientaciones sencillas, como primeras aproximaciones. Hay muchas otras formas de implementar estas estructuras, seguramente más eficientes. Por ejemplo, en lugar de devolver un dato se podría devolver un puntero, los parámetros podrían pasarse por referencia y ser constantes, de este modo aseguraríamos que no se modificarán los datos pasados, etc. También se podrían utilizar funciones virtuales para utilizar herencia.

Sin embargo, se eligió la manera más sencilla de implementar las estructuras para comenzar a asimilarlas sin desviar nuestra atención demasiado. La recomendación es que de a poco, las implementaciones que desarrollen, las vayan ampliando y mejorando hasta lograr implementaciones robustas y genéricas, sin perder de lado la eficiencia y la claridad en el código.

De todas maneras, se recalca que lo visto es teórico, ya que C++ tiene definidas varias clases en la librería STL (Standard Template Library), entre ellas está la clase *list* que ya provee todos los métodos para que sólo tengamos que usarlos.

## 8. Bibliografía

- Construcción de Software Orientado a Objetos (2da edición) – Bertrand Meyer (1999)
- Data Structures and Algorithms – Alfred Aho (1983)