UNIDAD I V – PILAS, COLAS, LISTAS

OBJETIVOS DE LA DECIMA SEMANA

Clase teórica

- INTRODUCCION.CONCEPTO DE PILA, COLA Y LISTA. CLASE NODO Y LISTA ENLAZADA DE NODOS. Implementando una clase Pila. Implementando una clase Cola.

Clase práctica

- class Nodo. Class ListEnl. Class Pila. Class Cola.

OBJETIVOS DE LA UNDECIMA SEMANA

Clase teórica

- Implementando una clase Lista. Enunciados combinando estructuras de datos.

Clase práctica

- Class Lista.

OBJETIVOS DE LA VIGESIMO SEGUNDA SEMANA

Clase teórica

- Enunciados combinando estructuras de datos.

Clase práctica

- class Enunc01.

Todo el material de esta unidad en el sitio del Laboratorio de Sistemas. (labsys.frc.utn.edu.ar)

Autor: Ing. Tymoschuk, Jorge
Coordinación: Ing. Guzmán, Analía

PILAS, COLAS y LISTAS

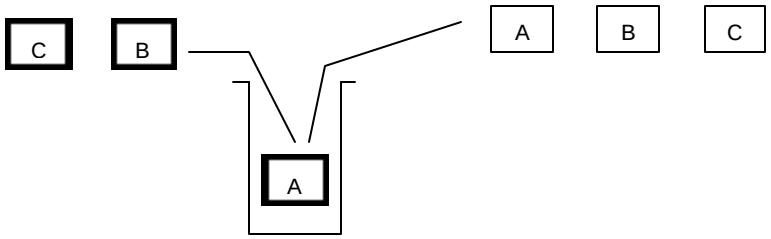
Introducción

En este capítulo vemos en detalle las estructuras de datos pilas, colas y listas.
Las dos primeras son estructuras de datos que almacenan y recuperan sus elementos atendiendo a un estricto orden. Las pilas se conocen también como estructuras **LIFO** *(Last in, First-out),* último en entrar-primero en salir), las colas como estructuras **FIFO** *(First in, First out),* primero en entrar-primero en salir.

CONCEPTO DE PILA

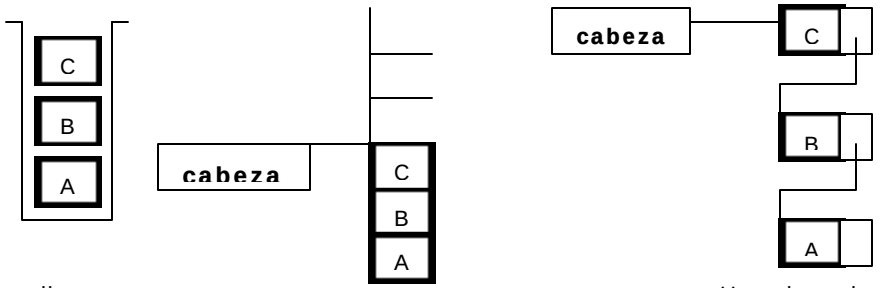
Una **pila (stack)** es una colección ordenada de elementos a los que solo se puede acceder por un único lugar o extremo. Los elementos de la pila se añaden o se quitan (borran) de la misma sólo por su parte superior (**cima).** Entonces**, Insertar y Excluir son las únicas operaciones lícitas (posibles) en una pila.**

Al operarse únicamente por la parte superior de la pila, al excluir elementos retiramos el que está en la cima, el último que incluimos. Si repetimos esta operación retiraremos el penúltimo, el antepenúltimo, o sea en orden inverso al que los incluimos. Supongamos que tenemos las letras 'A', 'B' y 'C', las incluimos y luego excluimos dos de ellas, necesariamente 'C' Y 'B'. Gráficamente:



La operación de **Insertar** sitúa un elemento dado en la cima de la pila y **Excluir** lo elimina o quita

El lenguaje C (o C++) no tiene sentencias específicas para tratamiento de pilas, a diferencia de los arrays, donde si tiene, y mucho. Para trabajar con pilas debemos "pedir prestado" recursos a otras estructuras. Se pueden implementar pilas en arrays, en archivos, con nodos vinculados por punteros. Gráficamente, si tenemos 'A', 'B', 'C' "apiladas":



La pila, Conceptualmente — Usando Array — Usando nodos vinculados

En la implementación con arrays, **cabeza** es el índice del último casillero ocupado, (o del primero disponible). Es simple, pero tiene el inconveniente de soportar una estructura típicamente dinámica (la pila) con una estática (el array): El array puede ser demasiado grande, o pequeño.

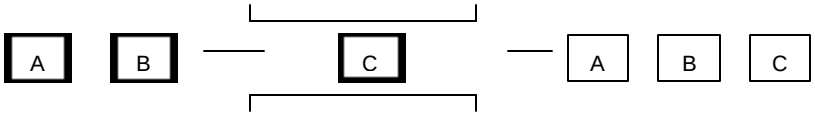
En la implementación con nodos vinculados **cabeza** es un puntero al primer nodo de esta estructura lineal, el cual contiene siempre el último elemento insertado. Su implementación requiere un poco mas de trabajo, pero al ser ambas estructuras del mismo tipo no existen los inconvenientes de la anterior. Por ello, esta es la que veremos en detalle.

La pila es una estructura importante en computación. Algunos de sus usos, transparentes para usuarios de un lenguaje de programación, es el retorno automático al punto en que una función fue invocada. Supongamos que main() invoca a func1(); dentro de func1() se invoca a func2(); dentro de func2() se invoca a func3(), allí procesamos y llegamos a la sentencia return. Una pila es la que contiene la información para que el retorno sea a un punto del cuerpo de func2(), luego de dirigirá nuestro retorno a func1(), y finalmente al main().

CONCEPTO DE COLA

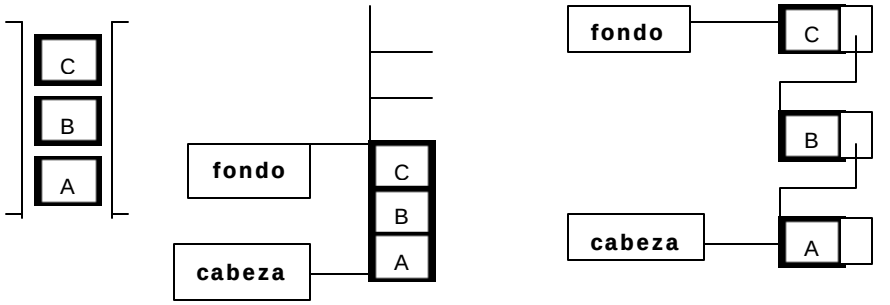
Una **cola** es una estructura de datos que almacena elementos en una fila (uno a continuación de otro). Cada elemento se inserta en la cola (parte final) de ¡a fila y se suprime o elimina por el frente (Cabeza). de la fila Las aplicaciones utilizan una cola para almacenar y liberar elementos en su orden de aparición o concurrencia. **Insertar y Excluir son las únicas operaciones lícitas (posibles) en una cola.** La única diferencia con la pila es que estas operaciones se realizan en extremos opuestos. **Insertar** por el fondo, **excluir** por el frente.

Entonces **Insertar y Excluir** se realizan en el mismo orden. Un buen ejemplo de cola es el de personas esperando utilizar el servicio público de transporte. (Si dijéramos un mal ejemplo también es verdad). La gestión de tareas de impresión en una única impresora de una red de computadoras es otro ejemplo. Supongamos que tenemos las letras 'A', 'B' y 'C', las incluimos y luego excluimos dos de ellas, necesariamente 'A' Y 'B'. Gráficamente:



frente (cabeza) fondo

Como en el caso anterior, el lenguaje C (o C++) tampoco tiene sentencias específicas para tratamiento de colas. Para trabajar con colas debemos "pedir prestado" recursos a otras estructuras. Se pueden implementar colas en arrays, en archivos, con nodos vinculados por punteros. Gráficamente, si tenemos 'A', 'B', 'C' "encoladas":



La cola, conceptualmente

Usando array

Usando nodos vinculados

En la implementación con arrays, **fondo** es el índice del último casillero ocupado, (o del primero disponible). Es simple, pero tiene dos inconvenientes:

- de soportar una estructura típicamente dinámica (la cola) con una estática (el array)
- al excluir elementos del array lo hacemos desde el lado de la cabeza, con lo que comienzan a existir casilleros libres al inicio del array. Para el array, la cabeza de la cola se desplaza hacia atrás. Este inconveniente se soluciona "simulando" que el arrray es circular, pero sigue vigente el primer inconveniente.

En la implementación con nodos vinculados **cabeza** es un puntero al primer nodo de esta estructura lineal, el cual contiene siempre el primer elemento insertado y **fondo** al último. Esta implementación requiere un poco mas de trabajo, pero al ser ambas estructuras del mismo tipo no existen los inconvenientes de usar arrays. Por ello, esta es la que veremos en detalle.

En general, la cola es la estructura de gestión de recursos de alguna manera escasos. Impresoras que atienden a un grupo de estaciones de trabajo en una red, tablas de bases de datos que varios usuarios pretenden actualizar en forma exclusiva, etc, etc.

CONCEPTO DE LISTA.

Una lista es una colección o secuencia de elementos dispuestos uno detrás de otro.
Una lista, por ejemplo la que hacemos para hacer las compras de la semana (quincena, mes) en el súper. Si somos ordenados y eficientes, hacemos esta lista de manera que en una sola recorrida, con un mínimo de tiempo y pasos, compramos todo lo anotado. Pero he aquí que llega nuestro cónyuge, ve nuestra lista, tacha las dos primeras líneas y las tres ultimas, luego incluye nuevas líneas en diferentes puntos de la lista, modifica otras líneas (esta marca es mas barata, solo esta sirve), etc. Una lista debe soportar un comportamiento de este tipo. De hecho, ni pensar en un array. (Si Ud. cree que es fácil, pruebe hacerlo...) En cambio, una estructura lineal de nodos

vinculados por punteros no tiene ningún problema. Es mas, es tan adecuada que se confunden los conceptos: la **lista de los items** que necesitamos comprar con el de **lista enlazada** de nodos donde vamos a implementar la de items.

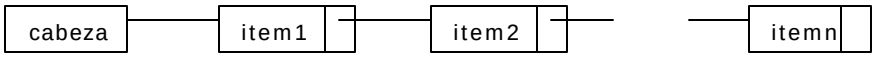
Una vez planteado que es lo que vamos a tratar (Pilas, Colas, Listas) y como (Implementándolas en estructuras lineales de nodos vinculados por punteros), definamos un par de puntos.

Nodo: Un nodo es una combinación de una parte de datos y un enlace (puntero) al siguiente nodo. Para la parte de datos, ya estamos hablando de items, nos viene bien usar la clase Items (Código/valor), que ya usamos en búsqueda y ordenamiento. Con lo cual, nuestra clase Nodo contendrá (relación **tiene un**) un objeto Item y un puntero al próximo nodo.

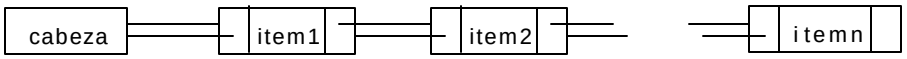
Lista enlazada: se compone de una serie de nodos enlazados mediante punteros.

Constructivamente, pueden ser de varios tipos.

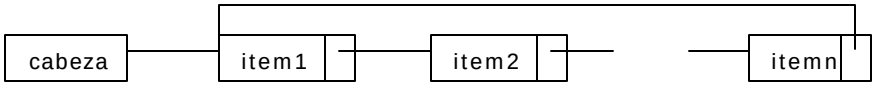
Lista simplemente enlazada: Cada nodo contiene un puntero que vincula al próximo. En el último, el puntero es nulo. Solo podemos avanzar. Gráficamente:



Lista doblemente enlazada: Cada nodo contiene dos enlaces, uno a su nodo predecesor y otro a su sucesor. Podemos avanzar y retroceder. Gráficamente:

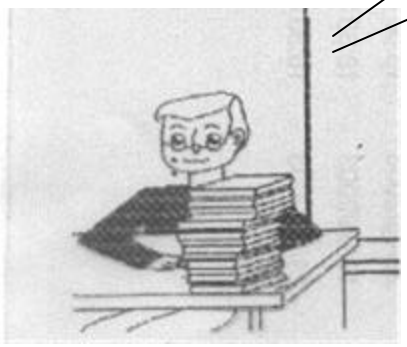


Lista circular simplemente enlazada: Variante de la lista simplemente enlazada, su último nodo enlaza al primero.

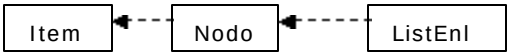


Por simplicidad, y porque raramente se necesita de otra cosa, usaremos la **Lista simplemente enlazada**.

Bueno, y como hacemos todo esto en objetos, profe ?



*Bueno, hay que ir viendo que relaciones tenemos. Ya dijimos que un Nodo **tiene un** Item. Y que una lista enlazada se compone, es decir, tiene nodos, de nuevo relación **tiene un.** Hasta aquí tendríamos:*



*La lista enlazada (class ListEnl) debe ser el **ancestro común** de Pila, Cola, Lista, para que estas clases cumplan la relación **es un** con ListEnl. Y que atributo común tienen todas ellas ? Sin duda, el puntero de entrada, puntero a nodo, **cabeza o frente**, como prefieran llamarlo ...*

La clase ListEnl es entonces el ancestro común, la clase base de Pila, Cola y Lista. Y que **comportamiento común** tienen todas ellas ?. Veamos un poco:

	Pila	Cola	Lista	Común
Recorrer nodos, procesando información contenida en ellos ?	no	no	si	**no**
Incluir nodos, ya sea al inicio, fin o intermedios ?	si	si	si	**si**
Excluir nodos, ya sea al inicio, fin o intermedios ?	si	si	si	**si**
Verificar si la lista está vacía ?	si	si	si	**si**

Vemos que hay bastante comportamiento común. Aunque lo de común, dependiendo del detalle con que se lo observe, puede no serlo tanto. Vamos a un caso concreto, sea el de **incluir nodos**.

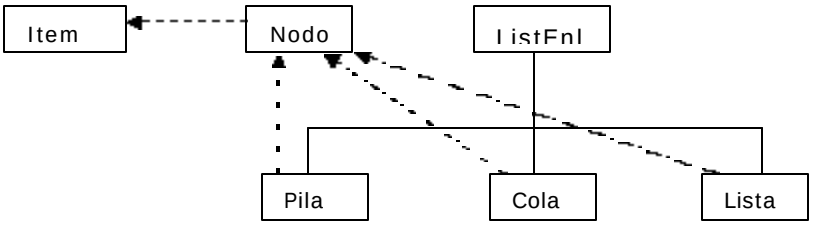
En la pila, incluimos al frente.
En la cola, incluimos al fondo.
En la lista, en cualquier lado. Vamos a suponer, en principio, que incluimos en orden de código, a menos que se especifique otra cosa.

Entonces tenemos que **incluir nodo** en cada caso es una cosa distinta, pero en todos los casos **se debe incluir.** Desde el punto de vista del programador, es mucho mejor si él puede codificar genéricamente **incluir(parámetros)** a especificar inclInicio(par), inclFin(par) o inclOrd(par). O sea, codificar siempre **incluir(par)** y que sea el lenguaje el que se ocupe de hacerlo al frente, fondo u ordenado según se trate de una pila, cola o lista. Y así con todas las acciones que haya que tratar. Es mejor trabajar con la **acción genérica incluir(par)**. Y que los detalles de su implementación dependan del objeto en la cual se implementa. Y para que se pueda trabajar bien con este concepto de **acción genérica** C++ aporte dos herramientas esenciales:

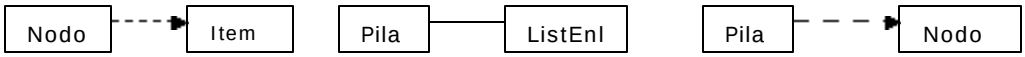
- ✓ **Sobrecarga.** Funciones con igual nombre, pero distinta "firma". Distinta firma significa un diferente juego de parámetros. Ej.: cálculo (int, int) tiene distinta firma que cálculo(int, float), cálculo(int, int, int), cálculo(int, char, int), etc.

- ✓ **Polimorfismo.** Funciones con igual nombre, igual firma, pero distinta implementación dependiendo del tipo de objeto invocante.

Por ahora presentamos los conceptos. Cuando entremos al detalle los ejemplificaremos. Entonces, siendo que nuestra **Lista Enlazada** tiene atributos y comportamiento común, sin duda es el ancestro que precisamos. Nuestra estructura de clases está quedando:



Las relaciones implementadas son:



Nodo **tiene un** Item Pila (Cola, Lista) **es un** ListEnl Pila **utiliza** Nodo (Colegas)

Veamos nuestra clase **Nodo.**

```
# include <G:\Borland\BC45\Ejemplos\AED2004\Unidad3\Items\Items.cpp>
class Nodo{                    //      Una clase de implementación de un Nodo
protected:                     // para una lista simplemente enlazada simple ...
    Item item;                         //  Nodo tiene un Item y
    Nodo * prox;                       //  un puntero al próximo Nodo...
public:
    Nodo(int, float, Nodo *);          // Un constructor con argumentos;
    Item getItem(){return item;}       // Retornamos el objeto item almacenado en Nodo
    void setItem(Item it){item = it;}; // Inicializamos idem
    Nodo * getProx(){return prox;}     // Retornamos puntero al proximo nodo
    void setProx(Nodo * ptr){prox = ptr;}// Inicilizamos puntero prox
    void        toString();            // Exhibimos el contenido de Nodo
    void implementar();
};

Nodo::Nodo(int cod=0, float val=0.0, Nodo * sig=NULL) {// Constructor del Nodo
    item.setCodigo(cod);
    item.setValor(val);
    prox = sig;
}

void Nodo::toString() {
    cout << "Datos del item ";
    item.toString();   // Usamos toString() de Item
    cout << ", prox " << prox << endl;
```

```
}

void Nodo::implementar(){
    cout << "Demo class Nodo \n";
    cout << "Visualizando objeto \n";
    toString();         // Usamos toString() de Nodo
    cout << "\nUna visión mas clara de los datos: \n";
    cout << "El código del item vale "<<item.getCodigo()<<endl;
    cout << "y su valor asociado es  "<<item.getValor()<<endl;
    cout << "Terminado.\n";
}

# include <G:\Borland\BC45\Ejemplos\AED2004\Unidad4\Nodo\Nodo.cpp>
void main(){
    Nodo nodo(10,20.75,NULL); // construimos un nodo
    nodo.implementar();
};
```

Lista nuestra **class Nodo**.

Vamos a **class ListEnl**, Lista Enlazada de Nodos.
Dijimos que esta clase es un **ancestro común**, un punto de partida para una estructura hereditaria de donde podamos heredar atributos y comportamiento.

Cual es el atributo común, imprescindible para poder trabajar con Pilas, Colas y Listas ? Simplemente el punto de entrada al la lista enlazada con la que las implementaremos. este punto de entrada es un puntero a nodo, lo definiremos entonces **Nodo * frente** y lo declararemos protected para que pueda ser heredado.

Y que comportamiento común tenemos ? **incluir, excluir, toString, implementar**. No podemos definirlos en **ListEnl,** cada clase heredera lo necesita distinto. Pero si nos conviene declararlo, y especificarlo **virtual** para notificar al lenguaje que estos métodos serán redefinidos.

C++ permite dos formas de declarar estos métodos virtuales:

- Declararlo como virtual puro: se **lo declara normalmente y se agrega =0. Ejemplo: virtual void mostrar()= 0;** una clase, al tener un método virtual se dice que es **abstracta**, no puede ser implementada, es decir no podemos instanciar, construir objetos de ella. No deja de ser un posible inconveniente.
- Declararlo como virtual, con un cuerpo de sentencias, que puede ser vacío o no. **Ejemplo: virtual void mostrar(){;}.** Su implementación es nula, mostrar no muestra nada, pero puede ser redefinido en clases herederas haciendo lo que sea necesario. Y la clase no es abstracta, puede ser instanciada.

En nuestro caso, el único atributo que tenemos, el objeto de la class ListEnl es el puntero **Nodo * frente** . Si optamos por la primera alternativa, deberemos instanciarlo en **cada una** de las clases derivadas. Si por la segunda, podemos hacerlo **directamente en ListEnl**. El autor se inclina por la segunda alternativa.

```
# include <G:\Borland\BC45\Ejemplos\AED2004\Unidad4\Nodo\Nodo.cpp>
class ListEnl{          //      Una clase base de una lista enlazada para
protected:              // implementar Pila, Cola, Lista  ...
    Nodo * frente;                  //  El puntero de entrada.
    virtual void incluir(int,float,int);
```

```
    virtual Nodo excluir(int);
    virtual void toString(){;}
public:
    ListEnl(){frente = NULL;}   //      Un constructor sin argumentos;
    int vacia(){return frente == NULL;}
    virtual void implementar(){;}
};

void ListEnl::incluir(int cod, float val,int trace=0){
    frente = new Nodo(cod,val,frente);
    if(trace)
        cout << "incluido cod " << cod << ", val " << val << endl;
}

/* paso a paso,  Pila::incluir() es lo siguiente:
new obtiene de la memoria dinámica un area del tamaño suficiente para albergar
un objeto Nodo.
Invocamos al constructor de Nodo, pasándole los parámetros entre ()
Ejecutamos el cuerpo del constructo r, esto nos inicializa el objeto item e
inicializa prox a la direccion contenida por frente.
La dirección del area obtenida por new es asignada a frente. Tenemos un nuevo
nodo que apunta al segundo (o a nadie si es el primer nodo que insertamos) */

Nodo ListEnl::excluir(int trace=0){
    Nodo nodo;                          // Construimos un objeto nodo inicializado con ceros.
    if(vacia()) return nodo; // Retornamos el objeto vacío
    Nodo * aux = frente;        // Guardamos en aux la dirección del primer nodo
    frente = frente->getProx(); // redirecionamos frente, saltando el primer nodo.
                                // para ello, asignamos a frente el resultado de conca-
                                // tenar los punteros frente->prox
    nodo = *aux;        // Guardamos contenidos del nodo antes de borrarlo
    delete(aux);        // Borramos el ex primer nodo;
    if (trace)
        cout << "excluido cod " << nodo.getItem().getCodigo()<< endl;
                                                                    // Será que anda ???
    return nodo;
}


/* Que significa nodo.getItem().getCodigo() ?
Lo que se necesita exhibir es el código del item del nodo.
Disponemos de modo, lo tenemos declarado como protected en esta clase.
No tenemos acceso a item, entonces lo obtenemos mediante la función que lo retorna, getItem()
Obtenido item, obtenemos codigo mediante getCodigo()
Complicado, es cierto. Hay otra forma, reduciendo el encapsulamiento mediante la declaración de
la clase ListEnl como amiga (friend) de Nodo y Nodo amiga de Item. Es otra forma ... */
```

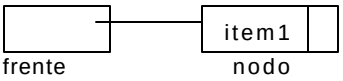
Que clase tan simple, verdad ? Si dijéramos que ahora solo resta heredar de ella personalizando cada método según se trate de Pila, Cola, Lista estamos diciendo la verdad, solo que no es tan simple... hay mucho detalle, vamos parte por parte...
En una Pila insertar() y excluir(), todo se hace por un único extremo. Y ese será el de frente, entonces Pila puede heredar todo esto de ListEnl ...
Cada nodo que insertemos deja al anterior atrás (mejor, abajo). Gráficamente:

Implementando class Pila

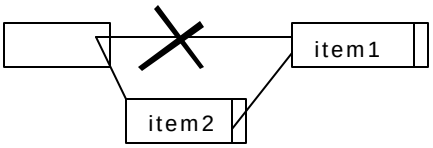
Inicialmente, al construir Pila,

frente

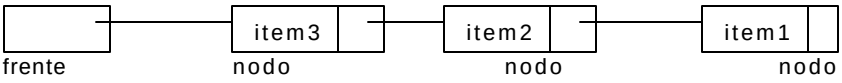
En la implementación, si insertamos un prime r nodo



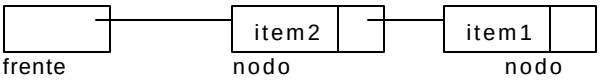
Si insertamos un segundo nodo, al inicio, es necesario un re direccionamiento de punteros.



Si seguimos insertando , nos queda



Si excluimos, estamos volviendo a una situación anterior




```
# include <G:\Borland\BC45\Ejemplos\AED2004\Unidad4\ListEnl\ListEnl.cpp>
class Pila: public ListEnl{ //  Implementando Pila a partir de ListEnl
protected:
    void toString();
public:
    Pila();                 //      Un constructor sin argumentos;
    void implementar();
};

Pila::Pila():ListEnl(){;}       // El constructor de ListEnl hace el trabajo

void Pila::toString() {
    cout << "Contenidos del objeto pila \n";
    cout << "Lamento, interior pila no visible \n";
    cout << "Unicamente incluir(), excluir()...\n";
}

void Pila::implementar(){
    cout << "Demo class Pila \n";
    for(int i=0;i<5;++i)incluir(i,(float)(i*2),1);
                        // Incluimos 5 nodos
    excluir(1);         // Excluimos un par
    excluir(1);
    cout << "Terminado !!!"<<endl;
}

# include <G:\Borland\BC45\Ejemplos\AED2004\Unidad4\Pila\Pila.cpp>
void main() {
    Pila pila;                  // construimos una pila
    pila.implementar();         // la implementamos
};
```

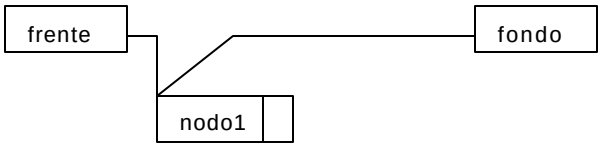
Implementando class Cola.

Ya hemos visto que en la cola los nodos se insertan por el fondo y se excluyen por el frente, como en la pila. Pareciera que **excluir()** lo podemos heredar de ListEnl y solo debemos redefinir **incluir().** Ocurre que para **incluir()** debemos antes recorrer la lista para posicionarnos en el último nodo, y si la lista es larga, esto es ineficiente. Es mejor la idea de tener también un puntero apuntando el último nodo. Esta optimización hace que **incluir()** sea distinta. Y cuando excluimos, siempre el primer nodo, y este llega a ser el último, frente y fondo deben ser NULL, entonces **excluir()** será también distinta. Conclusión: Si no optimizamos **excluir()** puede heredado de ListEnl, pero **incluir()** necesita ser redefinido(). Si optimizamos, debemos redefinir ambos.

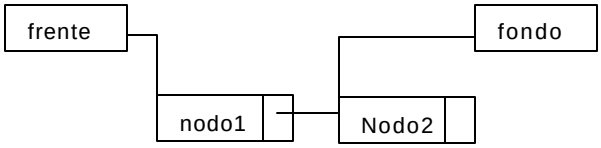
Graficamente:

Inicialmente, al construir Cola

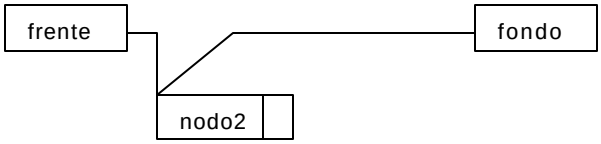
Al incluir el primer nodo



Al incluir el segundo



Al excluir() un nodo



La clase Cola entonces queda:

```
# include <G:\Borland\BC45\Ejemplos\AED2004\Unidad4\ListEnl\ListEnl.cpp>
class Cola: public ListEnl{ //  Implementando Cola a partir de ListEnl
protected:
    Nodo * fondo;
    void toString();
public:
    Cola();                 //      Un constructor sin argumentos;
    virtual void incluir(int,float,int);
    virtual Nodo excluir(int);
    void implementar();
};

Cola::Cola():ListEnl(){  // El constructor de ListEnl
    fondo = NULL;
}

void Cola::toString(){
    cout << "Contenidos del objeto Cola \n";
    cout << "Lamento, interior cola no visible \n";
```

```
    cout << "Unicamente incluir(), excluir()...\n";
}

void Cola::incluir (int cod, float val,int trace=0){
    Nodo *  aux = new Nodo(cod,val,NULL);
    if(fondo)    // ya teníamos nodos, redireccionar el último
        fondo->setProx(aux);  // el ex último se redirecciona al nuevo
    fondo = aux;         //  y fondo apunta al nuevo
    if(!frente)    //       no teníamos nodos, este es el primero
        frente = aux;
    if(trace)
        cout << "incluido cod " << cod << ", val " << val << endl;
}

Nodo Cola::excluir (int trace=0){
    Nodo nodo;                  // Construimos un objeto nodo inicializado con ceros.
    if(vacia()) return nodo; // Retornamos el objeto vacío
    Nodo * aux = frente;        // Guardamos en aux la dirección del primer nodo
    frente = frente->getProx(); // redirecionamos frente, saltando el primer nodo.
                                // para ello, asignamos a frente el resultado de conca-
                                // tenar los punteros frente->prox
    if(!frente)   // No quedan mas nodos
        fondo = NULL;
    nodo = *aux;        // Guardamos contenidos del nodo antes de borrarlo
    delete(aux);       // Borramos el ex primer nodo;
    if (trace)
        cout << "excluido cod " << nodo.getItem().getCodigo()<< endl;
                                                                    // Será que anda ???
    return nodo;
}

void Cola::implementar() {
    cout << "Demo class Cola \n";
    cout << "Incluimos 5 nodos \n";
    for(int i=0;i<5;++i)incluir(i,(float)(i*2),1);
    cout << "Ahora excluimos 2 \n";
    excluir(1);           // Excluimos un par
    excluir(1);
    cout << "Terminado !!!"<<endl;
}

# include <G:\Borland\BC45\Ejemplos\AED2004\Unidad4\Cola\Cola.cpp>
void main() {
    Cola cola;           // construimos una cola
    cola.implementar();  // implementamos su demo
};
```

```
(Inactive G:\BORLAND\BC45\EJEMPLOS\AED2004\UNIDAD4\COLA\MAINCOLA.EXE)
Demo class Cola
Incluimos 5 nodos
incluido cod 0, val 0
incluido cod 1, val 2
incluido cod 2, val 4
incluido cod 3, val 6
incluido cod 4, val 8
ahora excluimos 2
excluido cod 0
excluido cod 1
Terminado !!!
```

Implementando class Lista.

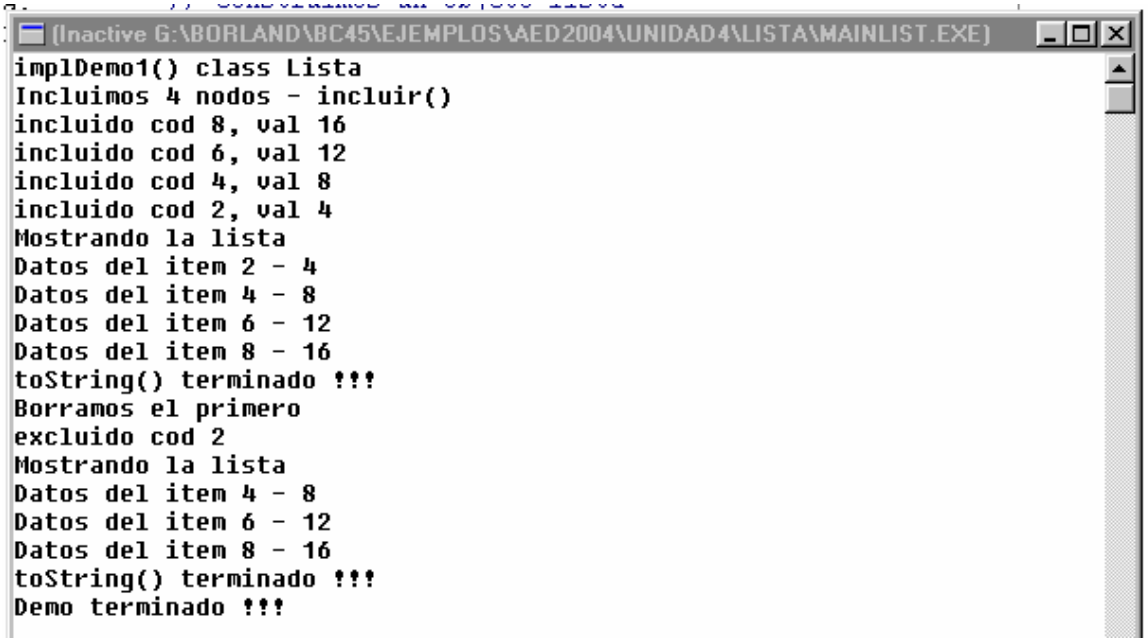
Dijimos que la lista tiene el comportamiento mucho más extenso. Varias formas de incluir, excluir, recorrer, buscar, etc. Podemos definirle tanto comportamiento como lo permita nuestra imaginación. Pero vamos a definir algo mínimo, algo así:

vacia(). Verifica si lista está vacia. Lo heredamos de ListEnl
incluir(int trace). Inserta nodos al inicio. Lo heredamos de ListEnl
excluir(int trace). Excluye nodos al inicio. Lo heredamos de ListEnl
inclOrd(Item item, int trace). Inserta nodos. Supone lista ordenada por código.
exclOrd(Item item , int trace). Excluye nodos. Supone lista ordenada por código.
actValor(Item item, int trace). Actualiza valor del nodo a partir del item parámetro
existe (**Item item , Nodo *).** Verifica si el ítem informado existe

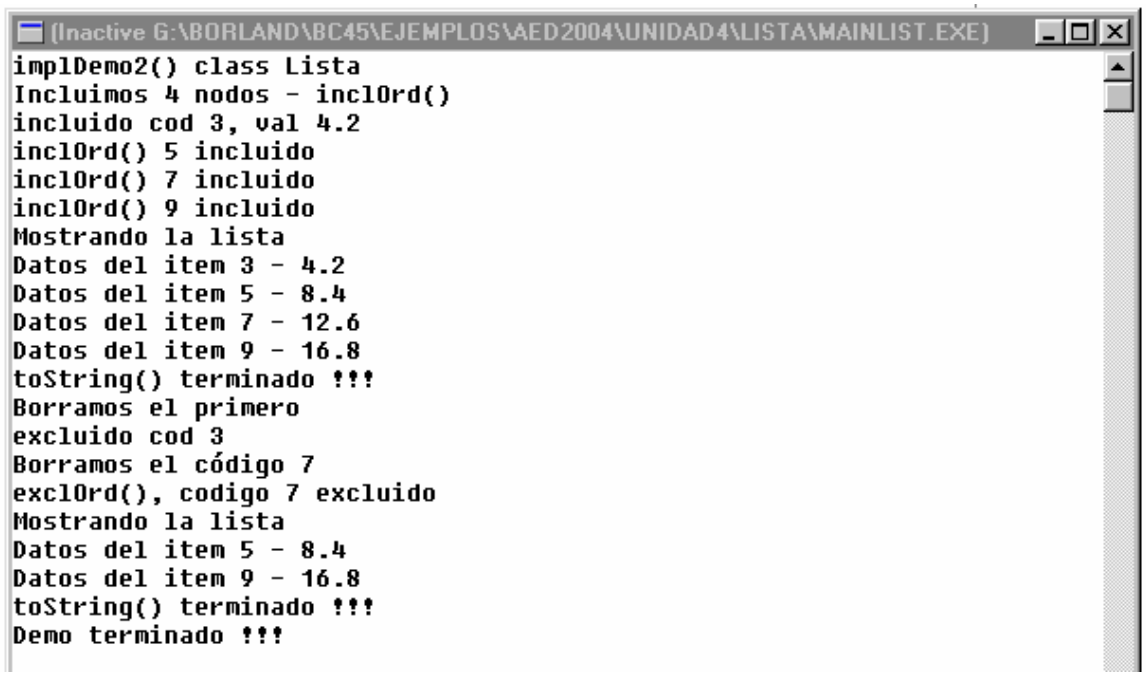
```
# include <G:\Borland\BC45\Ejemplos\AED2004\Unidad4\ListEnl\ListEnl.cpp>
class Lista : public ListEnl{ //  Implementando Lista a partir de ListEnl
protected:
    void toString();
    int existe(int, Nodo *&); // Verifica inexistencia del item parámetro.
public:
    Lista();                    //      Un constructor sin argumentos;
    ~Lista();                   // Destructor;
    void inclOrd(int, float, int); //Inserta nodos. Lista ordenada por código.
    Nodo exclOrd(int, int);     //Excluye nodos.Lista ordenada por código.
    void actValor(int, float, int); // Actualiza valor del item del nodo.
    void implDemo1();           // Un primer demo, como no es posible capturar todo
    void implDemo2();           // serán necesarios varios más ...
    void implDemo3();
    void implDemo4();
    void implDemo5();
};

Lista::Lista():ListEnl(){;}     // El constructor de ListEnl hace el trabajo

Lista::~Lista() {               // Un destructor
    Nodo *p=frente;
    while (p){
        frente=p;
        p=p->getProx();
        delete frente;
    }
    frente=NULL;
};

/* el método existe() tiene una doble funcionalidad:
    1) retorna
        1 si el código del item parámetro existe
        0 en caso contrario

    2) inicializa la referencia miPosic

        miPosic = NULL
                .la lista está vacia
                .el código del item parámetro es menor que el código del primer nodo

        miPosic = sombra (sombra es el puntero que sique "pegado" a ptr)
                . el código del ptr->nodo  es igual al código del item parámetro
                . el código del ptr->nodo  es mayor al código del item parámetro
                . el código del item parámetro supera al del último nodo        */

    int Lista::existe(int codPar, Nodo *&miPosic){
    int exis = 0;               // A priori, suponemos no existe
    Nodo * ptr = frente; // Puntero para recorrer la lista.
    Nodo * sombra = frente; // Puntero sombra, irá atras de ptr al recorrer lista
    int codNod;
    if (vacia())       //  La lista no tiene nodos
        miPosic = NULL;
    else{           //  Tenemos una lista con nodos
        for(;ptr;sombra = ptr, ptr = ptr->getProx()){
                codNod = ptr->getItem().getCodigo();
                if(codNod == codPar){ // hemos encontrado igual, basta de buscar ...
                        exis = 1; // existencia verdadera, pues el codigo encuentra igual ...
                        if(ptr == frente)       // ptr apunta al primer nodo,
                                miPosic = NULL; // código parámetro es == que el del 1er nodo
```

```
                    else                    // ptr apunta a un nodo intermedio o final,
                        miPosic = sombra;// Referenciamos la posicion del nodo anterior
                    break;
                }   // if(codNod == codPar)
                if(codNod > codPar){ // ya no vamos a encontrar igual, basta de nuevo
                    if(ptr == frente)       // ptr apunta al primer nodo,
                        miPosic = NULL; // código parámetro es < que el del 1er nodo
                    else                    // ptr apunta a un nodo intermedio o final,
                        miPosic = sombra;// Referenciamos la posicion del nodo anterior
                    break;
                }   // if(codNod > codPar
            } // for
            if(!ptr)  // Recorrimos lista completa, codPar es mayor al del último nodo
                    miPosic = sombra; //  Referenciamos la posicion del último nodo

        }   // else if (vacia())
        return exis;
}

void Lista::inclOrd(int codigo, float valor, int trace){
        Nodo * ptrNodo;
        int exist = existe(codigo, ptrNodo);
        if(exist){          // ya tenemos ese código, no podemos incluirlo nuevamente
            if(trace)cout << "inclOrd(), codigo " << codigo << " ya existe \n";
        }else{              // No existe el código del item parámetro, podemos incluir
            if(ptrNodo){    // es una inserción entre nodos o al fin de la lista
                    Nodo * aux = ptrNodo->getProx(); // obtenemos el puntero al próximo
                    ptrNodo->setProx(new Nodo(codigo,valor,aux));
                    if(trace)cout << "inclOrd() " << codigo << " incluido\n";
            }else           // Es una inserción al inicio de la lista o la lista está vacía
                    incluir(codigo,valor,1);// usamos el método adecuado, el heredado

        }   // else del if(exist)
}
/* que hace  ptr->setProx(new Nodo(codigo,valor,aux); ?
        vamos por partes
        Nodo(codigo,valor,aux) es la ejecución del constructor de Nodo que
        inicializa adecuadamente los atributos de valor y puntero del nuevo objeto
        nodo obtenido por new, cuya dirección es entregada a setProx, quien la usa
        para inicializar el atributo prox del nodo apuntado por el puntero sombra,
        con lo que queda adecuadamente vinculado el nuevo nodo.
        */
Nodo Lista::exclOrd(int codigo, int trace=0){
        Nodo * ptrNodo;             // para puntero al nodo
        int exist = existe(codigo, ptrNodo);  // Verificamos su existencia
        Nodo nodo;                  // Lo necesitamos para almacenar el nodo excluido
        if(exist)               // Tenemos igual codigo en lista, entonces podemos excluir
            if(ptrNodo){        // Debemos excluir un nodo intermedio o final
                    Nodo * ptrExcl = ptrNodo->getProx(); // puntero al nodo a excluir
                    nodo = *ptrExcl;                // almacenamos el nodo a excluir
                    Nodo * ptrProx =  ptrExcl->getProx(); // puntero al siguiente al excl.
                    ptrNodo->setProx(ptrProx);      // Vinculamos anterior al siguiente al excl.
                    delete(ptrExcl);                // Liberamos memoria del nodo excluido
                    if(trace)cout << "exclOrd(), codigo " << codigo << " excluido \n";
            }else                                   // El nodo a excluir es el primero
                    nodo = excluir(1);              // usamos el método adecuado, el heredado
        else        // No existe el item buscado
            if(trace)cout << "exclOrd(), codigo " << codigo << " no existe \n";
        return nodo;                                // retornamos el nodo excluido completo
}
void Lista::actValor(int codigo, float val, int trace=0){
        Nodo * ptrNodo;                             // para puntero al nodo
        int exist = existe(codigo, ptrNodo);      // Verificamos su existencia
        Nodo * ptrAct;              // Puntero al nodo cuyo valor de item actualizaremos
        if(exist){              // Tenemos código en lista, podemos actualizar valor asociado
            if(ptrNodo)     // Debemos actualizar en un nodo intermedio o final
                    ptrAct = ptrNodo->getProx(); // seteamos ptrAct con la dirección correcta
            else                    // El valor a actualizar está en el item del primer nodo
```

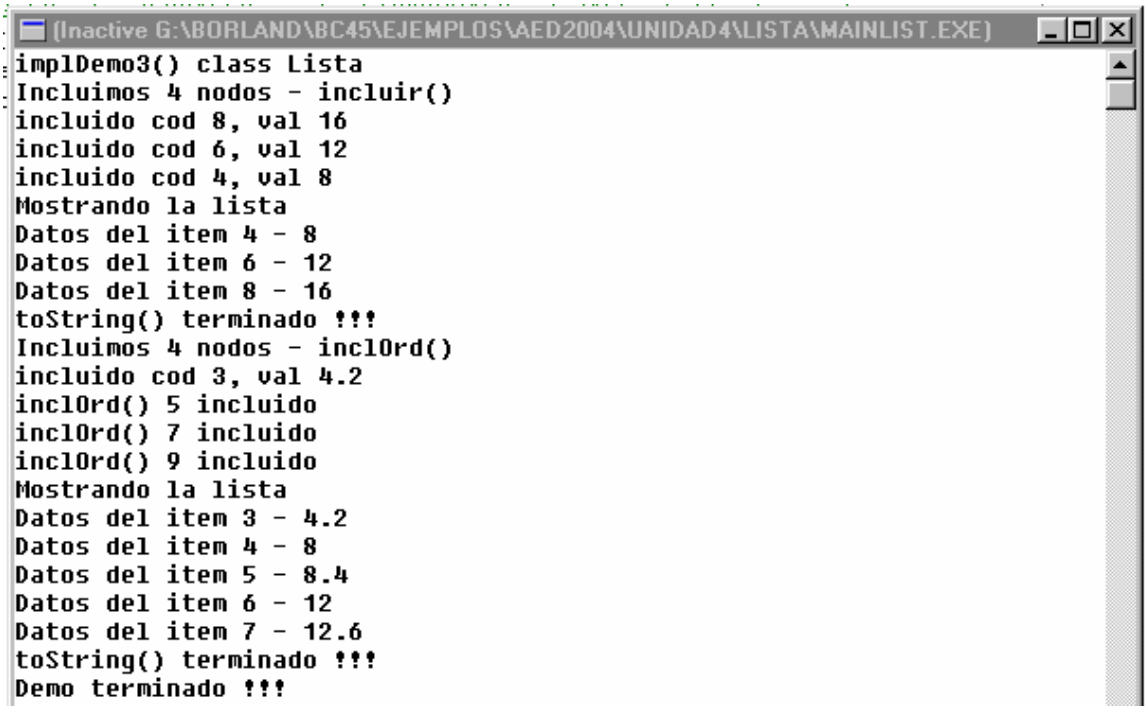
```
                ptrAct = frente;
        // En este punto, ptrAct apunta al nodo cuyo valor queremos actualizar
        Item it(codigo,val);            // Generamos un objeto item  y lo usamos para
        ptrAct->setItem(it);            // reemplazar el del nodo
        // if(trace)cout << "actValor(), código " << codigo << " a " << val << endl;
    }
    else                                // No existe el item buscado
        if(trace)cout << "actValor(), código " << codigo << " no existe\n";
}

void Lista::toString(){
    int cont = 0;  // Solo hemos de mostrar hasta 5 nodos
    if(vacia()) cout << "Lista vacia, nada a mostrar \n";
    else{               // Tenemos nodos, mostremos su contenido
        cout << "Mostrando la lista \n";
        for(Nodo * ptr = frente;ptr && cont < 5 ;ptr = ptr->getProx(),++cont)
                ptr->toString();
    }
    cout << "toString() terminado !!!  \n";
}


void Lista::implDemo1(){
    cout << "implDemo1() class Lista \n";
    cout << "Incluimos 4 nodos - incluir()\n";
    for(int i=4;i>0;--i)incluir(i*2,(float)(i*4),1);
    toString();
    cout << "Borramos el primero \n";
    excluir(1);
    toString();
    cout << "Demo terminado !!!"<<endl;
}
```




```
void Lista::implDemo2(){
    cout << "implDemo2() class Lista \n";
    cout << "Incluimos 4 nodos - inclOrd()\n";
    for(int i=1;i<5;++i)inclOrd(i*2+1,(float)(i*4.2),1);
    toString();
    cout << "Borramos el primero \n";
    excluir(1);
    cout << "Borramos el código 7 \n";
    exclOrd(7,1);
    toString();
    cout << "Demo terminado !!!"<<endl;
}
```

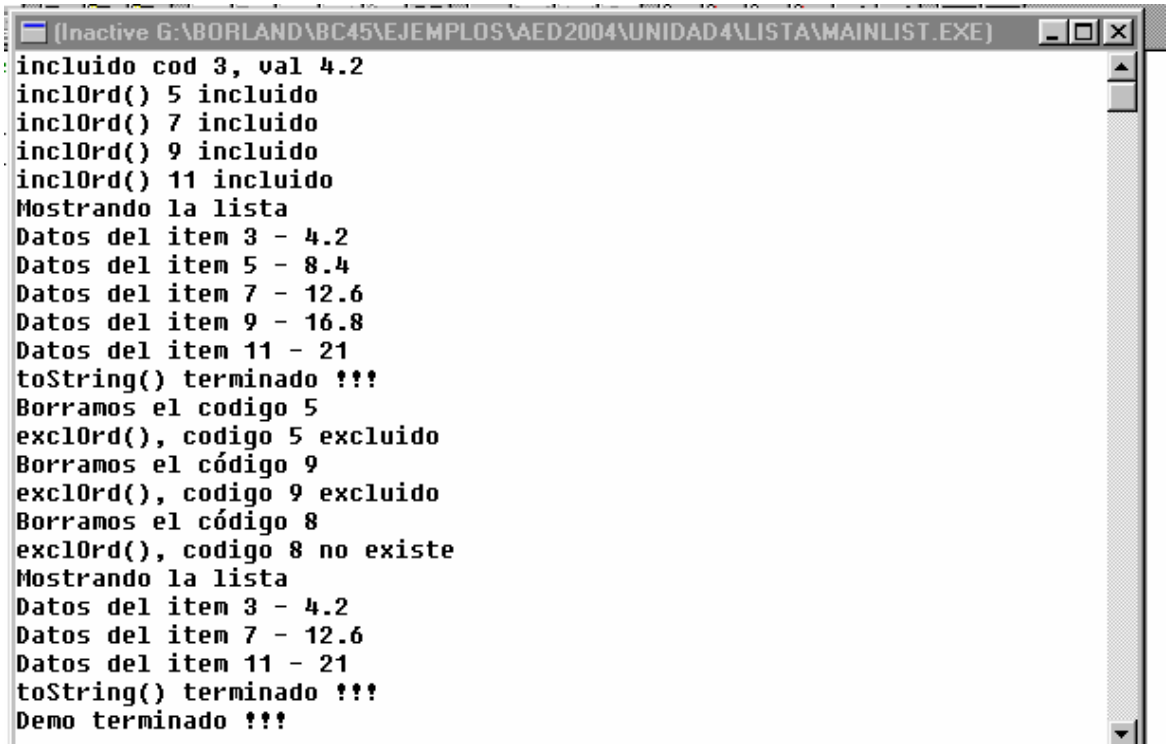



```
void Lista::implDemo3(){
    cout << "implDemo3() class Lista \n";
    cout << "Incluimos 4 nodos - incluir()\n";
    for(int i=4;i>1;--i)incluir(i*2,(float)(i*4),1);
    toString();
    cout << "Incluimos 4 nodos - inclOrd()\n";
    for(i=1;i<5;++i)inclOrd(i*2+1,(float)(i*4.2),1);
    toString();
    cout << "Demo terminado !!!"<<endl;
}
```




```
void Lista::implDemo4(){
    cout << "implDemo4() class Lista \n";
    cout << "Incluimos 5 nodos - inclOrd()\n";
```

```
    for(int i=1;i<6;++i)inclOrd(i*2+1,(float)(i*4.2),1);
    toString();
    cout << "Borramos el codigo 5 \n";
    exclOrd(5,1);
    cout << "Borramos el código 9 \n";
    exclOrd(9,1);
    cout << "Borramos el código 8 \n";
    exclOrd(8,1);
    toString();
    cout << "Demo terminado !!!"<<endl;
}
```



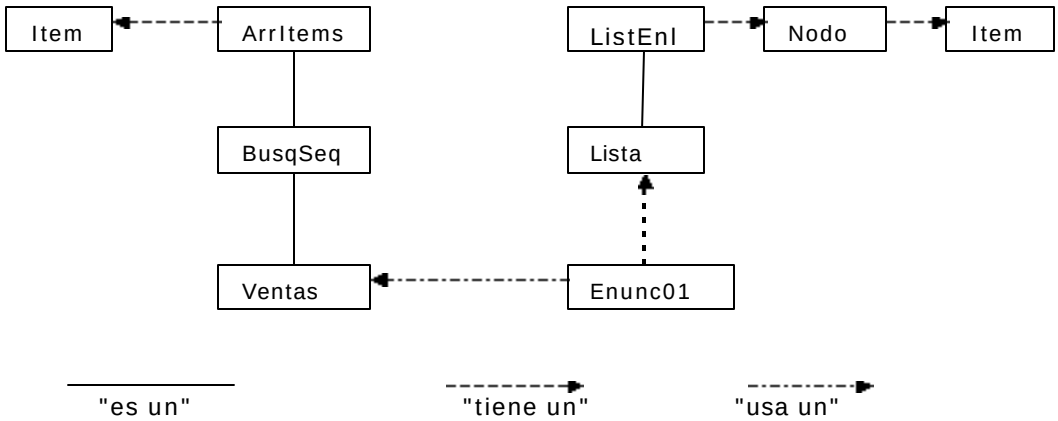

```
void Lista::implDemo5(){
    cout << "implDemo5() class Lista \n";
    cout << "Incluimos 3 nodos - inclOrd()\n";
    for(int i=1;i<4;++i)inclOrd(i*2+1,(float)(i*4.2),1);
    toString();
    cout << "Actualizamos el código 5 \n";
    actValor(5,10.3,1);
    cout << "Actualizamos el código 3 \n";
    actValor(3,20.6,1);
    cout << "Actualizamos el código 8 \n";
    actValor(8,20.6,1);
    toString();
    cout << "Demo terminado !!!"<<endl;
}
```

```
(Inactive G:\BORLAND\BC45\EJEMPLOS\AED2004\UNIDAD4\LISTA\MAINLIST.EXE)
implDemo5() class Lista
Incluimos 3 nodos - inclOrd()
incluido cod 3, val 4.2
inclOrd() 5 incluido
inclOrd() 7 incluido
Mostrando la lista
Datos del item 3 - 4.2
Datos del item 5 - 8.4
Datos del item 7 - 12.6
toString() terminado !!!
actValor(), código 5 a 10.3
actValor(), código 3 a 20.6
actValor(), código 8 no existe
Mostrando la lista
Datos del item 3 - 20.6
Datos del item 5 - 10.3
Datos del item 7 - 12.6
toString() terminado !!!
Demo terminado !!!
```

Todo está muy bonito, profe, pero como hacemos esos programas de antes, por ejemplo tenemos un array y debemos generar una lista con los códigos que ...

Me quieren poner en un aprieto, me parece ... Hum... veamos... tenemos 10 minutos ? Supongamos que tenemos las ventas de un comercio, las del viernes, las del sábado, cada una es un array de items código - valor. Dos arrays como entrada de datos. Y el tema podía ser detectar que se vende siempre, que se vende en fin de semana, algo así. Dos listas que se generan en el confronto de los arrays. Les parece que puede ser ? ...

El diagrama de clases:



/* **Enunciado 01**

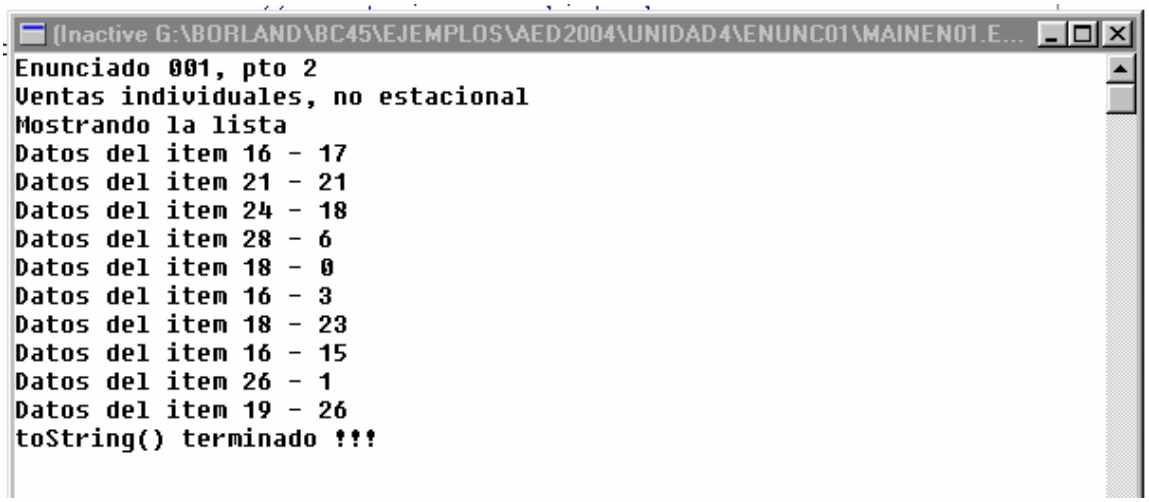
Se tienen dos objetos arrays de Items, class ArrItems
Ambos contiene las ventas de un comercio, en dos días sucesivos de una semana cualquiera.
El objeto vtasVier contiene las ventas del viernes.
El objeto vtasSab contiene las ventas del sábado.
Se pide:

```
        1) - Generar un objeto lista de lo vendido el sábado que no se vendió el
        viernes. Estamos queriendo detectar que es lo que se vende exclusivamente
        el fin de semana. Llamamos a este objeto vtasFSem. Queremos que en este
        objeto lista las ventas se acumulen por código.
        2) - Generar un objeto lista de lo vendido el sábado que tambien se vendió
        el viernes. Estamos queriendo detectar que es lo que se vende habitual
        mente, lo que la gente consume siempre. Llamamos a este objeto vtasEver.
        A diferencia del punto anterior, queremos que cada item del objeto vtasSab
        se incluya  en vtasEver directamente, sin acumulación.
    */

/* Estrategia de resolución.
    En principio, tenemos gran parte de comportamiento listo.
    En la parte de array de items, podemos usar virtual int existe(int);
    (Existe la clave parámetro ?, class BusqSec.)
    En la parte de listas, en class lista, tambien tenemos
    void inclOrd(int, float, int), (Inserta nodos. Lista ordenada por código. Se
    preocupa por la unicidad del codigo del item.)

    Evidentemente, es un problema que se resuelve con el concurso de varias
    clases. Para ordenar un poco esto, definimos que todo lo que haremos está
    dentro de una clase, naturalmente class Enunc01. Esta clase tendrá por lo
    menos dos métodos, que implementarán lo que pide en enunciado, que llamaremos
    punto01 y punto02.

    Y como relacionamos estas clases?  Comencemos por la salida, las listas.
    Nuestra class Enunc01 debe
        . heredar (Relación "es un"), de class Lista ?.
        . debe tener (Relación "tiene un") class Lista ?.
        . simplemente usar objetos lista ?

    Es evidente que class Enunc01 no es una especialización de Lista, entonces
    podríamos optar por cualquiera de las otras dos alternativas. Si suponemos
    que los objetos Lista que generamos en punto01 y punto02 serán usados por
    otros futuros métodos de esta clase o herederas, necesitamos de la relación
    "tiene un"

    La entrada de nuestro programa son los arrays de items. Una vez procesados,
    la información como la queremos está en las listas, no los necesitamos mas,
    entonces deberíamos deshacernos de ellos. Un buen lugar de definirlos es en
    punto01 y punto02. O sea que usaremos objetos arrays de items, de la
    clase derivada BusqSeq, porque precisaremos de este método

    Al trabajo.

    Nota: En la programación de este enunciado, se descubre que el atributo talle
    definido como protected en ArrItems, conteniendo la información del tamaño
    del array de items no esta disponible para la clase enunc01. Que hacemos ?
    Declaramos una clase Ventas que hereda publicamente de BusqSeq, que subsana
    este y algún otro inconveniente, y adelante ...
                */

# include <H:\Borland\BC45\Ejemplos\AED2004\Unidad3\Items\Items.cpp>
# include <H:\Borland\BC45\Ejemplos\AED2004\Unidad4\ventas\Ventas.cpp>
# include <H:\Borland\BC45\Ejemplos\AED2004\Unidad4\Lista\Lista.cpp>

class Enunc01{
    protected:
        Lista vtasFSem, vtasEver;
                /* Enunc01 tiene un objeto Lista, llamado ptrVen. Es un puntero a Nodo.
                        Como un puntero a Nodo ? Porque Lista hereda de ListEnl,
                        donde se lo define       Nodo * frente;                    */
        void punto01();
        void punto02();
    public:
        void implementar();
};
```

```
void Enunc01::punto01(){
        int codVSab;          // Codigo de un item vendido el sábado ...
        float valVSab;        // Y su valor
        int vendVie;                              // Vendido el viernes
        Nodo *ptrNodo;        // Lo necesita existe() de Lista
        int exist;                                        // Lo inicializa existe de lista al retornar
        Ventas vtasVie(10,'R'); // Generamos un array de 10 objetos item random
        Ventas vtasSab(30,'R'); // Generamos un array de 30 objetos item random
        int venSab = vtasSab.getTalle(); // cantidad de ventas del sábado
        Item  item;
        for(int i=0;i<venSab;++i){ // Recorremos esas ventas del sábado
                item = vtasSab.ptrItem[i].getItem(); // Obtenemos un item
                codVSab = item.getCodigo();    // ahora su código
                valVSab = item.getValor();     // y su valor
                vendVie = vtasVie.existe(codVSab); // Veamos las ventas del viernes
                if (!vendVie){   // Si no fue vendido el viernes, nos interesa ...
                        exist = vtasFSem.existe(codVSab,ptrNodo); // vemos si ya fué vendido
                        if (exist)                // Este código ya estaba en la lista.
                                vtasFSem.actValor(codVSab,valVSab);  //lo actualizamos
                        else                  // No estaba,
                                vtasFSem.inclOrd(codVSab,valVSab);  // lo incluimos
                }
        } // for
        cout << "Enunciado 001, pto 1 \n";
        cout << "Acumulado ventas fin de semana \n";
        vtasFSem.toString();
    }
```

```
(Inactive G:\BORLAND\BC45\EJEMPLOS\AED2004\UNIDAD4\ENUNC01\MAINEN01.E...
Enunciado 001, pto 1
Acumulado ventas fin de semana
Mostrando la lista
Datos del item 8 - 10
Datos del item 10 - 12
Datos del item 11 - 17
Datos del item 12 - 1
Datos del item 13 - 11
Datos del item 14 - 29
Datos del item 16 - 19
Datos del item 18 - 8
Datos del item 19 - 17
Datos del item 20 - 4
toString() terminado !!!
```

```
    void Enunc01::punto02(){
        int codVSab;          // Codigo de un item vendido el sábado ...
        float valVSab;        // Y su valor
        int vendVie;                              // Vendido el viernes
        Ventas vtasVie(30,'R'); // Generamos un array de 10 objetos item random
        Ventas vtasSab(30,'R'); // Generamos un array de 30 objetos item random
        int venSab = vtasSab.getTalle(); // cantidad de ventas del sábado
        Item  item;
        for(int i=0;i<venSab;++i){ // Recorremos esas ventas de sábado
                item = vtasSab.ptrItem[i].getItem(); // Obtenemos un item
                codVSab = item.getCodigo();    // ahora su código,
                valVSab = item.getValor();     // su valor
                vendVie = vtasVie.existe(codVSab); // Veamos las ventas del viernes
                if (vendVie)      // Si fue vendido el viernes nos interesa ...
                        vtasEver.incluir(codVSab,valVSab);  // lo incluimos
        } // for
        cout << "Enunciado 001, pto 2 \n";
        cout << "Ventas individuales, no estacional \n";
        vtasEver.toString();
    }
```




```
void Enunc01::implementar(){
    punto01();
    punto02();
}
```