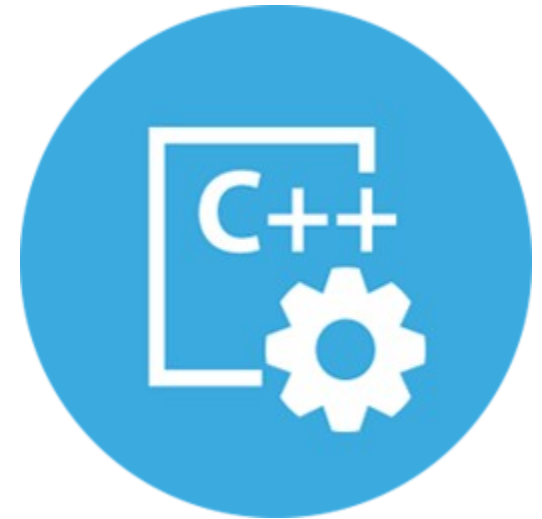




Universidad Nacional

Sede Regional Brunca

Estructuras de datos
M.C. Gabriel Núñez M.
III CICLO 2022

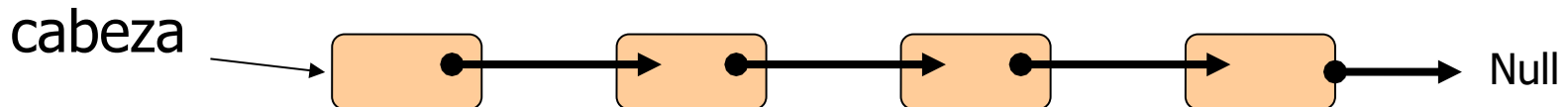


Tipos de estructuras de datos

- Las **listas** son comunes en la vida diaria:
 - **listas** de alumnos, **listas** de clientes, **listas** de espera, **listas** de distribución de correo, etc.
- Se utilizan frecuentemente en aplicaciones de recuperación de información, simulación, así como en algunas técnicas de administración de memoria, recursos, etc.
- Las **listas** son estructuras de datos muy útiles para los casos en los que se quiere almacenar información de la que no se conoce su tamaño con antelación.
- También son valiosas para las situaciones en las que el volumen de datos se puede incrementar o decrementar dinámicamente durante la ejecución del programa.

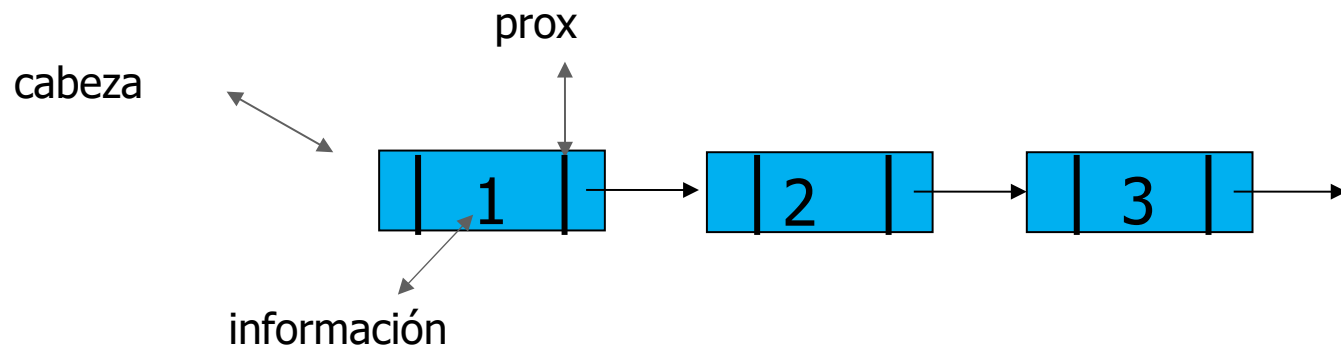
Listas

- Son estructuras de datos secuenciales de 0 o más elementos de un tipo dado, almacenados en memoria.
- Son estructuras lineales, donde cada elemento de la lista, excepto el primero, tiene un único predecesor y cada elemento de la lista, excepto el último, tiene un único sucesor.



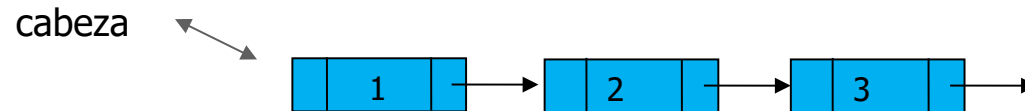
Estructura

- ❑ El número de elementos de la lista se llama longitud.
- ❑ Si tiene 0 elementos se llama **lista vacía**.
- ❑ En una lista podemos añadir nuevos elementos o suprimirlos en cualquier posición.

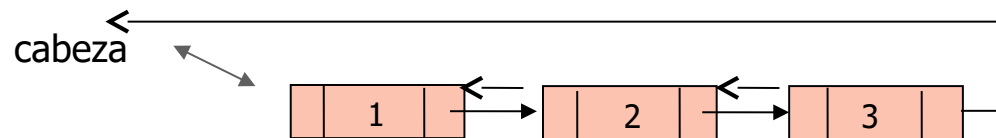


Tipos de listas

- ❑ No enlazada (arreglo)
- ❑ Enlazada (listas de objetos)
- ❑ Doblemente enlazada (anterior apunta al siguiente y viceversa)



- ❑ Circular (el último apunta al primero)



- ❑ A la vez pueden estar:
 - Ordenada
 - No ordenada

Características

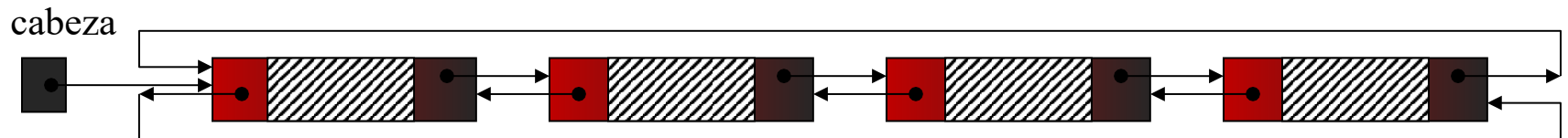
- ❑ Los elementos son accesibles y se pueden insertar y suprimir en cualquier posición de la lista.
- ❑ Las listas pueden concatenarse entre sí o dividirse en sub listas.
- ❑ Cuando aplicamos restricciones de acceso a las **listas** tenemos pilas y colas, que son **listas** especiales.
- ❑ Las listas son una estructura de datos más general que las colas y las pilas, por lo que se puede decir que éstas últimas son casos particulares de las listas.

Listas Enlazadas

- ❑ Secuencia “conectada” de nodos.
- ❑ Colección de elementos (denominados *nodos*)
- ❑ Tienen un nodo inicial (frente o cabeza) y un nodo final (cola).
- ❑ Cada nodo almacena (al menos) dos valores: un valor de la lista y un *puntero* o *referencia* que indica la posición del nodo que contiene el siguiente valor de la lista.
- ❑ Para que un nodo pueda acceder al siguiente y la lista no se *rompa* cada nodo tiene que tener un puntero que guarde la dirección de memoria que ocupa el siguiente elemento.
- ❑ Es necesario almacenar al menos la posición del primer elemento. (no se puede perder)
- ❑ Solo puede ser recorrida en secuencia.

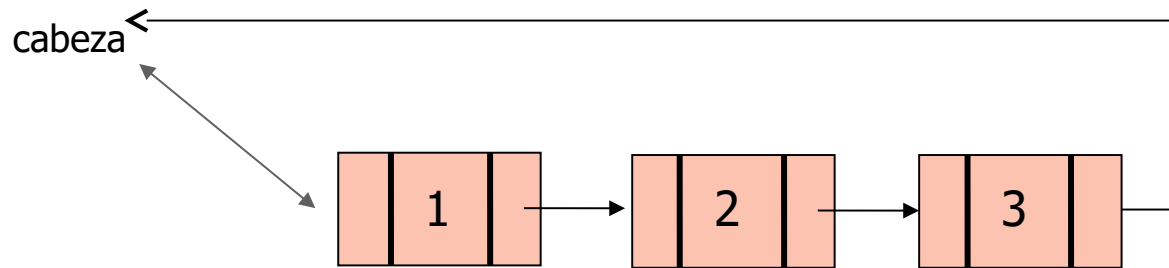
Lista doblemente enlazada

- ❑ Son **listas**, solo que agrega un puntero que apunta al nodo anterior.
- ❑ Pueden recorrerse en ambos sentidos, ya sea para efectuar una operación con cada elemento o para insertar, actualizar y borrar.
- ❑ Las búsquedas son más rápidas.
- ❑ Su inconveniente es que ocupan más memoria por nodo que una lista simple.



Listas circulares (enlazadas)

- Son **listas** en las que el último elemento tiene una referencia (enlace) con el primer elemento (cabecera).
- Pueden ser **listas** simples o doblemente enlazadas.



Operaciones sobre Listas

Algunas operaciones son:

- Inserción : insertar un elemento más a la lista (pila, cola)
- Borrado: eliminar un elemento de la lista (pila, cola)
- Recorrido: recorrer una lista
- Orden: ordenar una lista
- Búsqueda: buscar un elemento en la lista
- Consulta: consultar un dato de una lista

Listas (implementación)

- Para una lista de estudiantes se requiere:
 - Una clase: Clase Estudiante (objeto)
 - Clase Nodo, que contiene al objeto.
 - Clase Coleccion, estructura que administra los punteros y la lista en sí misma, ie. contiene los métodos que administran la lista de nodos.

UML

Estudiante

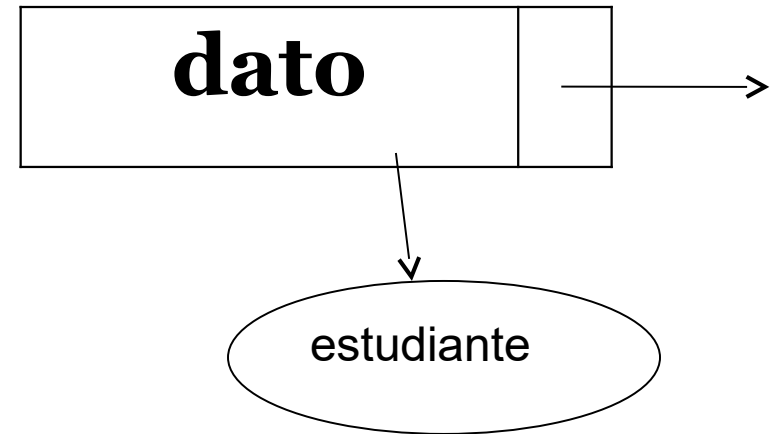
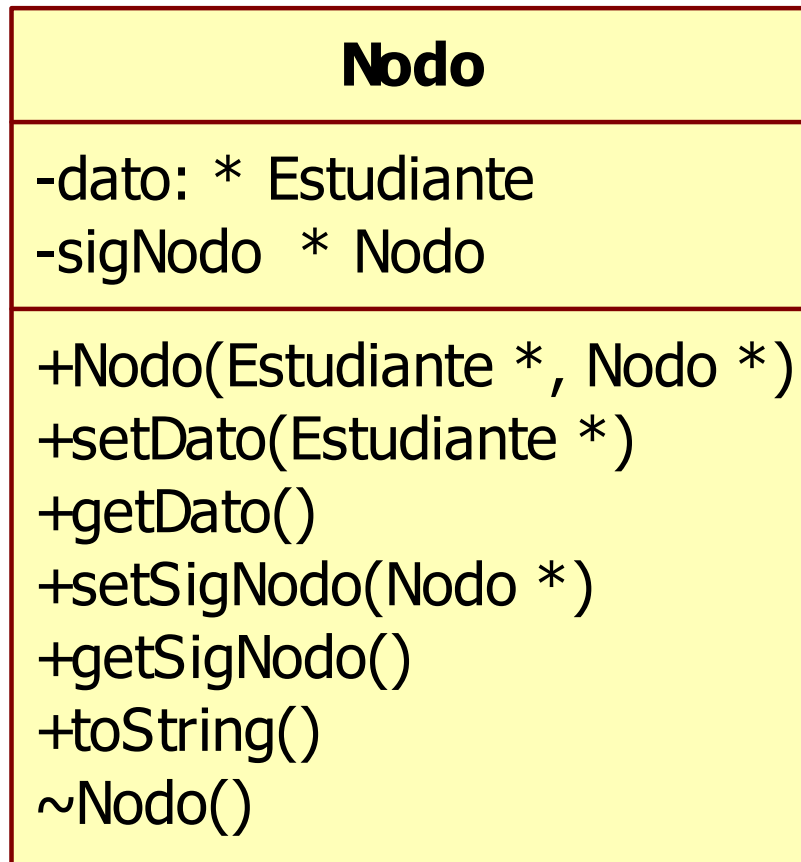
- cedula: string
- nombre: string
- apellido1: string
- apellido2: string
- promedio: float

+ Estudiante()
+ Estudiante(string, string,)
~ Estudiante
+ set.....
+ get.....
+ toString()
+ otros.....

clase Estudiante

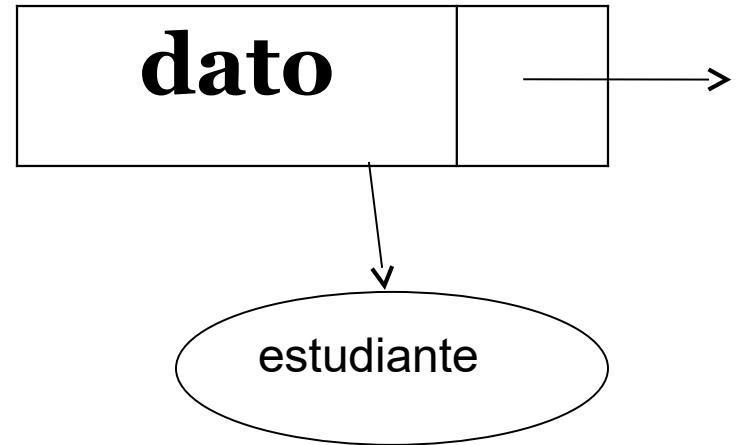
```
class Estudiante {  
private:  
    string cedula, nombre, apellido1, apellido2;  
    float promedio;  
public:  
    Estudiante (string id, string nom, float prom);  
    string getCedula();  
    void setCedula(string);  
    string getNombre();  
    void setNombre(string);  
    ....  
    string toString();  
};
```

Clase Nodo



Clase Nodo

```
#include "estudiante.h"
class Nodo{
private:
    Estudiante *dato;
    Nodo *sigNodo;
public:
    Nodo(Estudiante *, Nodo*);
    ~Nodo();
    void setData(Estudiante *ptrDato);
    Estudiante *getData();
    void setSigNodo( Nodo *sig) ;
    Nodo *getSigNodo();
    string toString();
};
```



Constructor y destructor

```
Nodo (Estudiante *ptrDato = NULL, Nodo *sig = NULL) {  
    dato = ptrDato;  
    sigNodo = sig;  
}  
  
~Nodo() {  
    delete dato; //único puntero  
}
```


Set y Get

```
void Nodo::setDato (Estudiante *ptrDato) {  
    dato = ptrDato;  
}  
Estudiante* Nodo::getDato ( ) {  
    return dato;  
}  
void Nodo::setSigNodo (Nodo *sig) {  
    sigNodo = sig;  
}  
Nodo * Nodo::getSigNodo () {  
    return sigNodo;  
}
```

toString()

```
string Nodo::toString() {  
    stringstream ss;  
    ss<< getDato()->toString()<<endl;  
    return ss.str();  
}
```

Clase Coleccion

Coleccion
- Nodo *actual - Nodo *primero
+ Coleccion() + ~Coleccion()

coleccion.h

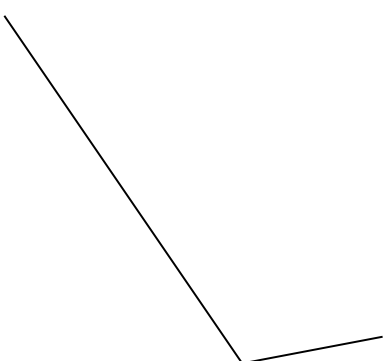
```
#include "nodo.h"
class Coleccion {
    private:
        Nodo *primero;
        Nodo *actual;
    public:
        Coleccion ( ) ;
        ~ Coleccion() ;
        void insertarPrimero (Estudiante *ptrDato );
        void borrarPrimero();
        string toString();
};
```

coleccion.cpp

```
Coleccion ( ) {  
    primero = actual = NULL;  
}
```

insertarPrimero ()

```
void insertarPrimero ( Estudiante *ptrDato) {  
    actual = primero;  
    if (primero == NULL ) //Lista Vacía  
        primero = new Nodo (ptrDato, NULL);  
    else  
        primero = new Nodo (ptrDato, actual);  
}
```



Observe que ambas
instrucciones son parecidas
con algunas consideraciones

insertarPrimero ()

```
void insertarPrimero ( Estudiante *ptrDato) {  
    primero = new Nodo (ptrDato, primero);  
}
```

borrarPrimero()

```
void borrarPrimero() { //Busca y elimina el primer nodo
    actual = primero;
    if (primero != NULL) {
        primero = actual->getSigNodo();
        delete actual;
    }
}
```


toString()

```
string Coleccion::toString() {  
    stringstream ss;  
    actual = primero;  
    ss << "Contenido de la lista:"<<endl;  
    while (actual != NULL){    //primero no se mueve  
        ss << actual->toString()<<endl;  
        actual = actual->getSigNodo();  
    }  
    return ss.str();  
}
```

Destructor

```
~Coleccion() {  
    while (primero != NULL ) {  
        actual = primero;  
        primero = primero->getSigNodo();  
        delete actual;  
    }  
}
```

insertarFinal()

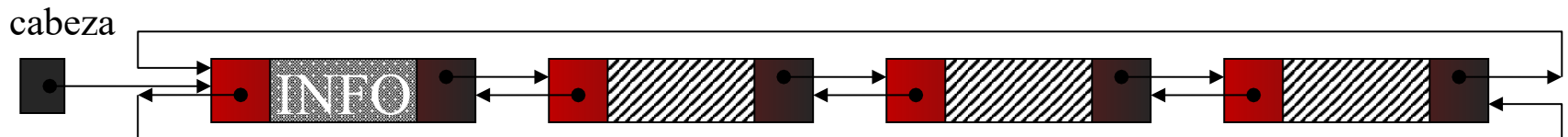
```
void insertarFinal ( Estudiante * ptrDato ) {  
    if (primero == NULL) { //Lista Vacía  
        primero = new Nodo (ptrDato , primero); }  
    else {  
        actual = primero;  
        while (actual->getSigNodo() != NULL ) { // al final  
            actual = actual->getSigNodo() ;  
        }  
        actual->setSigNodo ( new Nodo ( ptrDato, NULL) );  
    }  
}
```

borrarFinal()

```
void Coleccion::borrarFinal(){ //Busca y elimina el último nodo
    actual = primero;
    Nodo *anterior = actual;
    if (primero != NULL ){
        if (primero->getSigNodo()==NULL ) {
            delete primero;
            primero = NULL; }
        else {
            while (actual->getSigNodo() != NULL ) {
                anterior = actual;
                actual = actual->getSigNodo();
            }
            anterior->setSigNodo(NULL );
            delete actual;
        }
    }
}
```

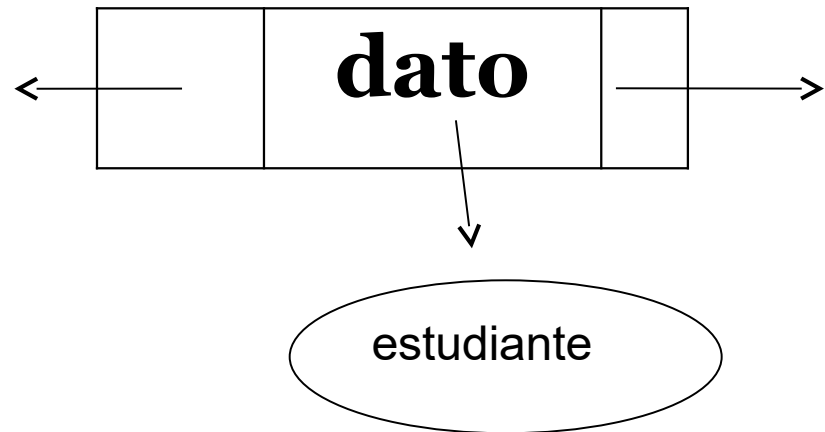
Lista doblemente enlazada

- ❑ Son **listas**, solo que agrega un puntero que apunta al nodo anterior.
- ❑ Pueden recorrerse en ambos sentidos, ya sea para efectuar una operación con cada elemento o para insertar, actualizar y borrar.
- ❑ Las búsquedas son más rápidas.
- ❑ Su inconveniente es que ocupan más memoria por nodo que una lista simple.



Clase Nodo

Nodo
-antNodo * Nodo -dato: * Estudiante -sigNodo * Nodo
+Nodo(Estudiante *, Nodo *) +setDato(Estudiante *) +getDato() +setAntNodo(Nodo *) +getAntNodo(Nodo *) +setSigNodo(Nodo *) +getSigNodo() +toString() ~Nodo()



Clase Nodo

```
#include "Estudiante.h"
class Nodo{
private:
    Nodo *antNodo;
    Estudiante *dato;
    Nodo *sigNodo;
public:
    Nodo(Estudiante *, Nodo*);
    ~Nodo();
    void setDato(Estudiante *ptrDato );
    Estudiante *getDato();
    void setAntNodo(Nodo *sig) ;
    Nodo *getAntNodo();
    void setSigNodo(Nodo *sig) ;
    Nodo *getSigNodo();
    string toString();
};
```

```
Nodo::Nodo (Nodo *ant = NULL, Estudiante *ptrDato = NULL, Nodo *sig = NULL) {  
    antNodo = ant;  
    dato = ptrDato ;  
    sigNodo = sig;  
}  
Nodo::~~Nodo() {  
    delete dato;  
}  
void Nodo::setDato (Estudiante *ptrDato) {  
    dato = ptrDato;  
}
```

```
Estudiante* Nodo::getDato ( ) {  
    return dato;  
}  
void Nodo::setAntNodo (Nodo *ant) {  
    antNodo = ant;  
}  
Nodo* Nodo::getAntNodo ( ) {  
    return antNodo;  
}  
void Nodo::setSigNodo ( Nodo *sig) {  
    sigNodo = sig;  
}  
Nodo* Nodo::getSigNodo ( ) {  
    return sigNodo;  
}
```

Coleccion Lista

ContenedorLista

-primero: * Nodo

+ContenedorLista()

+longitud()

+insertarPrimero()

+insertarFinal()

+borrarFinal()

+borrarPrimero()

+toString()

~ContenedorLista()

Métodos de la coleccion

```
void Coleccion::insertarPrimero ( Estudiante *ptrDato) {  
    primero= new Nodo (NULL,ptrDato, primero);  
}
```

```
void Coleccion::borrarPrimero() { //Busca y elimina el primer nodo  
    actual = primero;  
    if (primero !=NULL) {  
        primero= actual->getSigNodo();  
        primero->setAntNodo( NULL);  
        delete actual;  
    }  
}
```

insertarFinal()

```
void Coleccion::insertarFinal ( Estudiante * ptrDato ) {  
    actual = primero;  
    //Lista Vacía  
    if (primero==NULL) {  
        primero= new Nodo (NULL, ptrDato , primero); }  
    else {  
        actual = primero;  
        while (actual->getSigNodo() != NULL ) { // al final  
            actual = actual->getSigNodo() ;  
        }  
        actual->setSigNodo ( new Nodo (actual, ptrDato, NULL) );  
    }  
}
```

borrarFinal()

```
void Coleccion::borrarFinal ( ) {
    if (primero != NULL)
        if (primero->getSigNodo() == NULL ) {
            delete primero;
            primero = NULL;
        }
    else {
        actual = primero;
        while (actual->getSigNodo () != NULL ) { // al final
            actual = actual->getSigNodo() ;
        }
        delete actual->getSigNodo();
        actual->setSigNodo ( NULL) ;
    }
}
```

Ejercicio 2a

- ❑ Insertar ordenado, por el promedio del estudiante.
- ❑ Borrar un estudiante específico.

Actividad 2b

- ❑ ~Coleccion() : destruya la lista
- ❑ void reemplazarProm (string id, float prom):
reemplaza el promedio de un id específico.

Bibliografía

- [Booch96] Booch, Grandy;“ Análisis y Diseño Orientado a Objetos con Aplicaciones ”., Editorial Addison Wesley Logman, 2 da edición, México, 1996.
- [Martin94] Martín, James; Odell, James J;“Análisis y Diseño Orientado a Objetos”, Editorial Prentice Hall, 1era edición, México, 1994.
- [Rodríguez97] Rodríguez Rojas, Oldemar;“ *C ++ para ambientes gráficos* ”., Editorial Tecnológica de Costa Rica, 1era edición, Costa Rica, 1997.

□