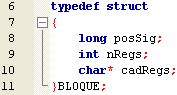
|  |  |  |
| --- | --- | --- |
| Práctica 2b | 23 de mayo  2010 | |
| Autor: José Ángel García Fernández | | Organización y Gestión de Archivos |

Memoria Práctica 2b

# Diseño y tipos de datos

He usado los ya creado en las prácticas anteriores, añadiendo alguna funcionalidad más que en un primer momento parecía que no hacía falta, pero al final si era necesaria, como por ejemplo añadirle la **sobrecarga de operadores** **>** y **<** al tipo **Registro** para facilitar la comparación y ordenación de los bloques. Destacar que muchos de los métodos de la clase original **ESFicheroBinario** hacían referencia a registros, ahora en la clase del archivo de bloques también se llamarán así debido a la herencia, pero no trabajaremos con los registros sino con el bloque completo, simplemente usando el tamaño del bloque y no el del registro.

### Tipo para representar el bloque

Declaramos un **struct** que contiene los campos del Bloque**.** Modelamos la posición del siguiente bloque en la lista como un **long,** el numero de registros como un **int** y la cadena que contendrá a los registros como un **char\***, sin tamaño predefinido ya que viene impuesta por el parámetro **n** (factor de blocaje).

### Clase BloqueIND



Esta clase envolverá a la estructura **BLOQUE** añadiéndole las funcionalidades que necesitaremos. Destacar que declaramos amigas a **INDICEND** y a **FileBloques** para poder acceder a la propiedad bloque.

## Métodos BloqueIND

### Añadir registros al bloque

## C:\C++\OGA\Practicas\Practica2b\imgs\addRegBIND.jpg

Este método permite introducir un nuevo registro en el bloque, usando el método **EstablecerCadenaRegistro**() y teniendo en cuenta si es el primer registro o no, devolvemos la longitud actual del bloque.

### Buscar registro en bloque

## C:\C++\OGA\Practicas\Practica2b\imgs\buscarBIND.jpg

Con este método buscamos en el bloque el registro con clave valor, para ello hacemos uso de la función Tokenize que partirá el bloque en **subcadenas** que representarán cada registro, para que en *[492,501]* podamos ir buscando el registro que cumpla la condición, si lo encontramos lo mostramos.

## Clase INDICEND

### C:\C++\OGA\Practicas\Practica2b\imgs\claseINDICEND.jpg

### Esta clase representa al índice no denso, desciende de ESFicheroBinario que fue creado en la primera práctica, y añadimos las mismas propiedades que se uso en la práctica 2.

## Métodos INDICEND

### C:\C++\OGA\Practicas\Practica2b\imgs\creaIndiceND.jpgGeneración Archivo de Índices

Este método se encarga de la creación del **archivo de índices** a partir del archivo de bloques ya creado. El método ira leyendo cada bloque *[38,51]*. Si no hay problemas obtengo el último registro del bloque *[43]* e inicializo las propiedades del registro de **índice primario** *[44,45]*, posteriormente lo escribo *[48*] y finalmente actualizo el **numero de registros insertados**, devolviendo este número *[52,53]*.

### Listado Ordenado de Archivo de Bloques

## C:\C++\OGA\Practicas\Practica2b\imgs\listarOrdenadoFileBloques.jpg

Este método se encarga de realizar la lectura ordenada del **archivo de bloques**, para ello va leyendo los registros del archivo de **índice no denso** *[73]* y posteriormente leyendo el bloque usando la posición asociada al índice *[74].* Para cada bloque lo muestro y además realizo el recorrido por la lista enlazada en la **zona de overflow**, mostrando los bloques que vamos consiguiendo *[78,83].* Finalmente devuelvo la variable i que valdrá el número de bloques mostrados.

### Lectura registro de indice

### C:\C++\OGA\Practicas\Practica2b\imgs\lecturaRIP.jpg

Son los mismo que se usaron en la practica 2, adaptados para la clase INDICEND. Uno sin especificación de posición y otro en la que se especifica en el que se llama al anterior.

### C:\C++\OGA\Practicas\Practica2b\imgs\BBindice.jpgBúsqueda Binaria en Archivo Índice

Con este método conseguimos saber donde se encuentra un registro con un determinado valor de **clave primaria**. Es una **búsqueda binaria** común pero ya que se trata de **un índice no denso** hay a que añadirle algunas peculiaridades. En la variable **pos** guardaremos la posición del registro de índice primario correcto y en **regIP** el propio registro. Cuando leemos un registro del índice *[98]*, comprobamos si este es mayor o igual que la clave buscada *[99]*, si se cumple esto, leemos el anterior *[102]* y si se cumple que la clave es mayor que la del registro leído **auxIP** *[106],*entonces**regIP** es el que apunta al bloque correcto. Si el bloque que le corresponde es el primero entrara en el **if** de la línea *[103]* porque dará un error al leer. Si no se cumple ninguna de las anteriores llegara a *[112]* en la que restringimos la búsqueda a la parte inferior. En las líneas *[114,125]* se hace lo mismo pero al revés. Finalmente en *[129]* devolvemos o la posición del registro o la posición del bloque, en función del parámetro **posEnIndice**.

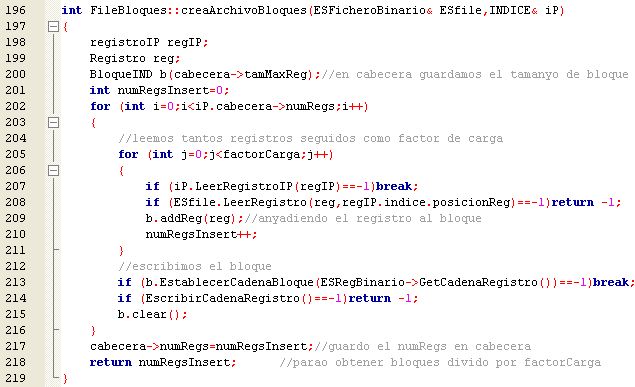
## Clase FileBloques

### C:\C++\OGA\Practicas\Practica2b\imgs\claseFileBloques.jpg

Esta clase modela al archivo de bloques, desciende también de **ESFicheroBinario**, y le añadimos la propiedad factor de Carga (será el factor de Blocaje) y el tamaño máximo de registro, la hacemos amiga de INDICEND para que esta pueda acceder a sus propiedades. Explicaremos más adelante alguno de sus métodos más importantes.

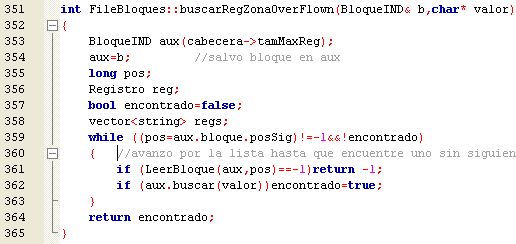
## Métodos FileBloques

### Generación Archivo de Bloques



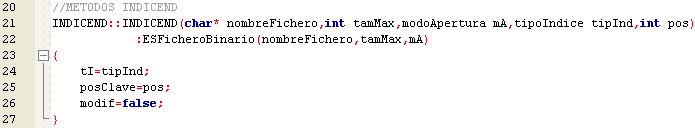
Este método será el encargado de realizar la construcción del **archivo de bloques**, para ello nos servimos del fichero de índice primario y del archivo de datos original, declaramos los registros y el bloque que usaremos en las líneas *[198,200].* Leeremos tantos registros como nos indique el **índice primario**, y dentro realizamos otro bucle que se repetirá tantas veces como registros queramos tener en cada bloque .En cada iteración leemos del fichero de índices y luego del fichero de datos y añadimos el registro al bloque *[205,211]*. Después escribimos el bloque completo en el archivo y lo resteamos *[213,215]*. Finalmente establecemos el número de registros insertado y lo devolvemos. Destacar que devolvemos el número de registros porque si nos hace falta el número de bloques bastaría con dividir este número por el **factor de blocaje**.

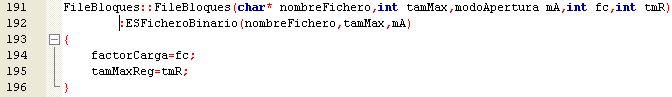
### BusquedaEnZonaOverflow

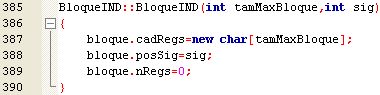


Este método realiza la búsqueda en la zona de **overflow** de un registro con un determinado valor de clave partiendo de un bloque inicial. Se va siguiendo la lista enlazada que establece la propiedad **posSig** del bloque y cada vez que tenemos uno valido hacemos uso del método buscar del bloque. Devolvemos encontrado.

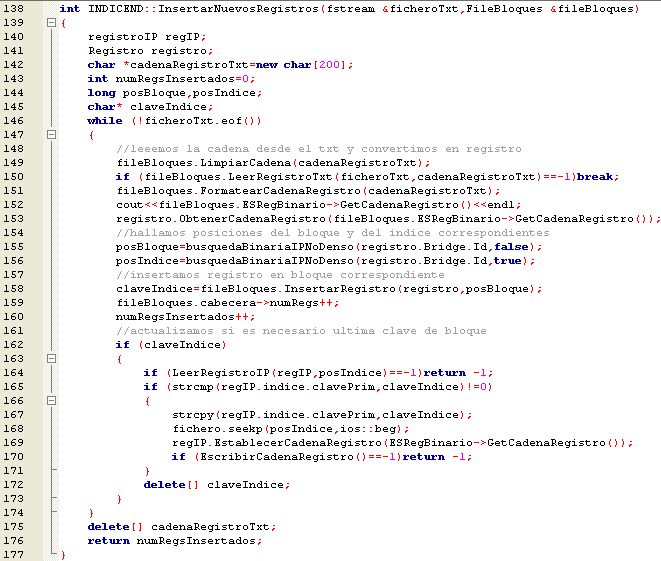
## Constructores

El constructor de las clases que heredan de **ESFicheroBinario** simplemente lo llaman siguiendo la sintaxis adecuada y luego inicializan sus propias propiedades, el de la case **BloqueIND** reserva espacio para la cadena de registros e inicializa nRegs a 0 y el siguiente a -1.





## Operación de Inserción

La operación de inserción interviene todas las clases creadas, empezaremos por el método base que pertenece a la clase INDICEND.

Este método será el que se llamara desde el **programa principal**, usara el archivo de bloques y el fichero de texto con los nuevos registros. Ira leyendo en bucle cada cadena del fichero *[146,174],* convirtiéndola a registro *[153]* y hallando la posición del bloque y del registro de índice que le corresponde *[155,156].* Tras esto usamos el método de la clase **FileBloques** para insertar un registro y actualizamos la **clave del índice** *[162,173]* en el caso de que sea necesario. Finalmente devolvemos el **número de registros insertados**.

### Insercion de Registro

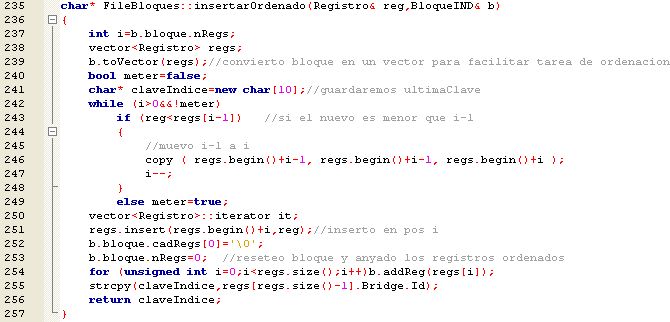
Este método es la base de la inserción de un registro en el fichero de bloques. Primero leemos el bloque *[256]* y comprobamos si está lleno *[257].* Si no lo está **insertamos ordenado** el registro en el bloque *[258]* y lo escribimos en el archivo *[259,261]* devolviendo la última clave *[262].* Si está lleno obtenemos el último bloque de la lista *[265,268]* y el último registro *[269]*. De nuevo comprobamos si está lleno [270], si no, actuamos como antes [271,275], si está lleno, comprobamos si el **nuevo** registro es menor que el **último** [278], si es menor eliminamos el **último** [279], insertamos el **nuevo** [280] e insertamos el **último** en zona de **overflow** *[281]*, si es mayor el **nuevo** registro *[282]* lo insertamos en la zona de **overflow** *[283]*, devolviendo en ambos casos la ultima clave.



### C:\C++\OGA\Practicas\Practica2b\imgs\insertZonOverF.jpgInserción de Registro en Overflow

Este método se encarga de introducir el **nuevo** registro en la zona de **overflown** enlazando los bloques. Primero avanzamos por la **lista de bloques** buscando uno que no tenga siguiente o que no esté lleno *[264,268].* Si está lleno y no tiene siguiente creamos un bloque nuevo añadiendo el **nuevo** registro y escribiendo este bloque en el archivo *[269,275],* a continuación **actualizamos el enlace del bloque** escribiéndolo en el archivo y actualizando la clave índice *[270,281]*. Si no está lleno **insertamos ordenado** el registro y escribimos el bloque *[283,290]*.

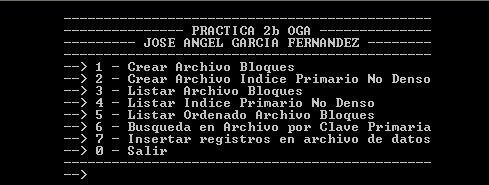
### Inserción Ordenada



Este método **inserta ordenado** un registro en el bloque. Para ello modelamos el bloque como un vector mediante el método de la clase **BloqueIND**  **tovector**() [239], y realizamos el algoritmo de la **inserción ordenada** *[242,251].* Tras realizar los desplazamientos correspondiente e insertar el registro en el vector, **reseteamos el bloque** *[252,253]* y lo creamos de nuevo añadiendo los nuevos registros a partir del vector ordenado *[254].* Finalmente devolvemos la **última clave del bloque** *[255,256].*

## Programa Principal creaNODENSO

La mayoría de las peculiaridades del programa se encuentran explicadas en el propio código, dejo aquí una imagen del programa principal como muestra. Destacar que antes de nada el programa pide el **factor de blocaje** que se usará los bloques para que se mantenga constante durante toda la ejecución.



# Material Adicional

Información encontrada en:

* Transparencias de clase.
* Guión Práctica
* Libro de Folk
* IDE Codeblocks (para descargarlo pulse en el tercer enlace)
* Webs
  + <http://www.cplusplus.com/>
  + <http://www.conclase.net/>
  + <http://www.codeblocks.org/downloads/5>

# Dificultades encontradas

Para hacer el método **toVector()** tuve una cantidad de problemas enorme relacionadas con el tratamiento de las cadenas c, al final opte por usar **string** que no me dio ningún problema.

Los problemas típicos de **flags** de los ficheros han vuelto a dar problemas, ya que algunas veces se quedaban activadas aunque cerrábamos el archivo, con lo que teníamos que hacer un .**clear**() para resetearlas.

El método de **busquedaBinaria**() también ha sido bastante complicado, ya que lo he tenido que modificar bastante desde el de la practica2.

Relativas a la **inserción** he tenido también muchos problemas, porque en un principio pensé en juntarlo todo en un método pero conforme iba creciendo la complejidad tuve que ir dividiendo el algoritmo para que fuera más fácil de implementar.

# Notas

Para abrir los proyectos use el **IDE Codeblocks** citado anteriormente.

* Archivos:
  + Bridges.dat
    - Archivo de registros apilado
  + BridgesB.dat
    - Archivo de registros en bloques
  + bridgesORIGINAL.dat
    - Archivo de registros original para uso en caso de reseteo
  + IndiceP.dat
    - Archivo de índice primario
  + IndiceND.dat
    - Archivo de índices no denso