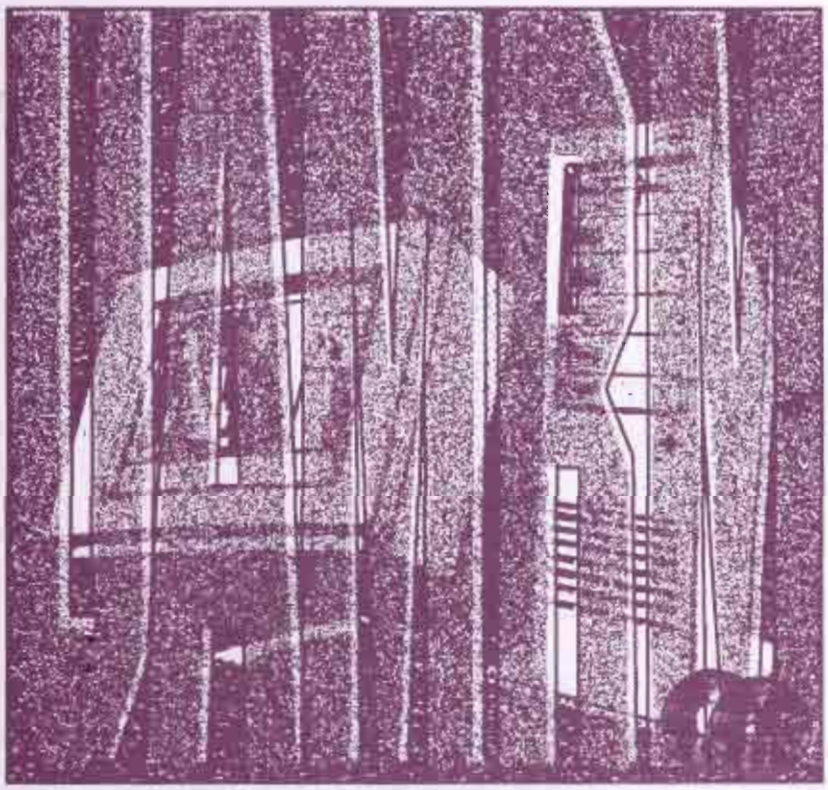
Manejador de archivos UAMRM

Memoria técnica de desarrollo

Hugo Moncayo López
Georgii Khachaturov



UNIVERSIDAD AUTÓNOMA METROPOLITANA
Casa abierta al tiempo Azcapotzalco

Maestría en Ciencias de la Computación

Manejador de archivos UAMRM

Memoria técnica de desarrollo

Manejador de archivos UAMRM

Memoria técnica de desarrollo

Este material fue dictaminado y aprobado por el Consejo Editorial de la División de Ciencias Básicas e Ingeniería, el 14 de febrero de 2001.

Autor: Hugo Moncayo López
Asesor: Georgii Khachaturov

Maestría en Ciencias de la Computación

Manejador de archivos UAMRM

Memoria técnica de desarrollo

Hugo Moncayo López
Georgii Khachaturov

UNIVERSIDAD AUTONOMA METROPOLITANA
Casa abierta al tiempo Azcapotzalco

División de Ciencias Básicas e Ingeniería
Departamento de Sistemas

UAM-AZCAPOTZALCO
RECTORA
Mtra. Mónica de la Garza Malo
SECRETARIO
Lic. Guillermo Ejea Mendoza
COORDINADOR DE EXTENSIÓN UNIVERSITARIA
Lic. Enrique López Aguilar
JEFA DE LA SECCIÓN DE PRODUCCIÓN Y DISTRIBUCIÓN EDITORIALES
Lic. Silvia Lona Perales

ISBN: 970-654-843-2

Corrección:
Marisela Júarez Capistrán
Ilustración de Portada:
Consuelo Quiroz Reyes
Diseño de Portada:
Modesto Serrano Ramírez

Universidad Autónoma Metropolitana
Unidad Azcapotzalco

Av. San Pablo 180, Col. Reynosa Tamaulipas
Deleg. Azcapotzalco, C.P. 02200
México, D.F.

Sección de producción
y distribución editoriales
tel. 5318-9222/9223. Fax 5318-9222

1a. edición, 2001

Impreso en México.

Tabla de Contenido

1. Introducción

Este reporte contiene la descripción del manejador de archivos UAMRM (UAM Record Manager) desarrollado como infraestructura para el almacenamiento de información visual necesaria para el proyecto "Un canal de aprendizaje visual para robot" dirigido por George Khachaturov[1].

UAMRM no pretende ser una aportación novedosa respecto a las técnicas de almacenamiento y recuperación de información. En su diseño se han utilizado técnicas que han demostrado su eficacia práctica para la implementación de manejadores de bases de datos comerciales.

El objetivo principal del proyecto es la creación de la infraestructura básica para soportar el almacenamiento y recuperación de información visual del sistema de visión de un robot con las siguientes características:

a). Fácil de adaptarse a nuevos requerimientos del proyecto. Para esto se necesita contar con los fuentes debidamente documentados.

b). Transportable con facilidad a cualquier plataforma de desarrollo.

c). Eficiente en cuanto uso de recursos, principalmente uso de memoria principal y aprovechamiento del espacio en disco.

d). Rápido en sus mecanismos de acceso.

e). Fácil de usar. Que cuente con un conjunto de operaciones básicas bien definidas y documentadas a fin de que sus usuarios no tengan ningún problema al utilizarlo.

2. Antecedentes

El objetivo del proyecto "Desarrollo de un canal de aprendizaje visual para robot" [Khachaturov 1] es lograr que un robot aprenda a reconocer los objetos de su espacio de trabajo mediante un proceso de aprendizaje en el que es instruido por un operador. El operador le muestra al robot los objetos del mundo real y le señala sus diferentes detalles. El robot extrae la información relativa al objeto de las imágenes que percibe y la almacena de manera conveniente en una base de datos. Posteriormente el robot tratará de identificar nuevos objetos que se le presenten mediante un proceso de reconocimiento de patrones en el cual se extraen los elementos de la imagen y se comparan con la información almacenada en la base de datos.

3. Justificación

Los manejadores de bases de datos comerciales están orientados a las aplicaciones de negocios tradicionales. Proporcionan una gran cantidad de servicios para ambientes transaccionales y que no resultan de utilidad práctica en el proyecto "Desarrollo de un Canal de Aprendizaje Visual para Robots" como son: el control de acceso concurrente, control para la integridad de transacciones, mecanismos de seguridad de acceso y otros. Estas opciones provocan que los productos comerciales resulten complejos, que requieran mayores recursos de hardware para su operación y que tengan procedimientos de instalación difíciles de aplicar. En casos como de ORACLE, SYBASE o INFORMIX es necesario los productos son específicos para un nivel de sistema operativo incluyendo los parches correspondientes; esto normalmente es responsabilidad del administrador del equipo y escapa al control de los investigadores. Otros tipos de paquetes como DBASE o ACCESS funcionan solamente en el ambiente de PC´s. Lo cual provocaría problemas al tratar de transportar la aplicación a un equipo UNIX.

En el aspecto económico, las licencias de las bases de datos comerciales se adquieren para una determinada plataforma de trabajo, y si se requiere trabajar en varios equipos y diferentes plataformas es necesario adquirir una licencia para cada equipo. Por otro lado, las interfaces con lenguajes de programación se mercadean como productos independientes y por lo tanto sería necesario adquirir licencias adicionales.

Por las razones antes mencionadas se decidió la elaboración de un producto que contuviera exclusivamente la funcionalidad necesaria para el proyecto de robótica, que pudiera ser instalado en cualquiera de los equipos disponibles para el proyecto, que resulte transportable a cualquier sistema operativo como unix, windows, o msdos, y sobre todo, que se cuente con los fuentes para poder adaptarlo a los nuevos requerimientos del proyecto.

4. Bases de Datos

Los sistemas para el manejo de bases de datos, por su arquitectura general se pueden clasificar en tres modelos principales: el modelo de red, el modelo jerárquico y el modelo relacional. A continuación describiremos las principales características de los tres modelos y analizaremos su pertinencia en el proyecto de robótica.

4.1 El modelo relacional

En el modelo relacional la información se representa en tablas llamadas relaciones. Una relación se define de la manera siguiente [8]:

> *Dada una serie de conjuntos D_1, D_2, ..., D_n (no necesariamente distintos) se dice que R es una relación sobre estos n conjuntos si es un conjunto de tuplas ordenadas $<d_1$, d_2, ..., $d_n>$ tales que $d_1 \in D_1$, $d_2 \in D_2$,.., $d_n \in D_n$. Los conjuntos D_1, D_2, ..., D_n son los dominios de R.*

Podemos visualizar una relación como la tabla que se muestra en la figura 1:

Código	Descripción	U. Medida	Precio Unit.	Exist
A22E123	Tornillo 1/4"	pza	0.75	300
A22E124	Tuerca 1/4"	pza	0.60	100
A22E125	Arandela	pza	0.20	200

Figura 1. Ejemplo de una tabla o relación.

El encabezado de la tabla contiene los nombres de los dominios con los que se va a poblar cada una de las columnas. Por ejemplo: en la primera columna los valores que aparecen son tomados del conjunto de los códigos de los artículos, los valores de la segunda columna se toman del conjunto de las descripciones de las partes y así sucesivamente. Cada renglón de la tabla es una tupla, y contiene los atributos que representan un objeto en particular.

La teoría de las bases de datos relacionales establece un conjunto de formas normales basadas en: la atomicidad de los dominios, la dependencia funcional, la dependencia multivaluada y la dependencia de reunión[8]. Las formas normales integran un marco teórico sólido para fundamentar un buen diseño de las bases de datos.

Una base de datos relacional es un conjunto de relaciones (tablas) entre las cuales no existen ligas físicas, es decir, las tablas se definen con un alto grado de independencia. La única restricción que se define entre tablas es la regla de integridad referencial aplicable a las llaves foráneas [8]. Esto se traduce en que el diseño de la base de datos no depende de la forma en que se va a explotar la información y por lo tanto la base de datos puede resultar apropiada para soportar las aplicaciones actuales y futuras.

En las bases de datos relacionales, la manipulación de los datos se realiza con un conjunto de operaciones básicas definidas en el álgebra relacional [8]. Cada operación del álgebra relacional tiene operandos que son relaciones y como resultado se obtienen nuevas relaciones. Entre las operaciones básicas podemos citar la selección, la proyección y la reunión. Esto ha permitido contar con SQL (Structured Query Language), un lenguaje de tipo declarativo basado en el cálculo de predicados de primer orden en el cual el usuario de la base de datos expresa las características de la información que desea obtener y el sistema determina la estrategia de acceso para obtenerla.

El modelo relacional ha resultado ideal para las aplicaciones llamadas comerciales, sin embargo, presenta problemas para el modelado de objetos complejos. Una de las características principales de las bases de datos relacionales es que no se maneja el concepto de estado, es decir, no existen apuntadores que durante su procesamiento nos indiquen cual es el último registro procesado para que futuros accesos puedan realizarse a partir de allí. En la aplicación que nos ocupa, la posibilidad de navegar a través de los registros de información es sumamente importante, ya que al tratar de reconocer un patrón visual es

necesario iniciar una búsqueda por algún criterio de semejanza y posteriormente recorrer las estructuras que satisfacen algún criterio de vecindad.

4.2 El modelo jerárquico

En el modelo jerárquico la información se representa mediante un conjunto de estructuras jerárquicas en forma de árboles. En cada árbol los nodos representan archivos y las aristas la liga de un nivel de jerarquía al siguiente. En la figura 2 se muestra un diagrama jerárquico para representar un sistema de capacitación:

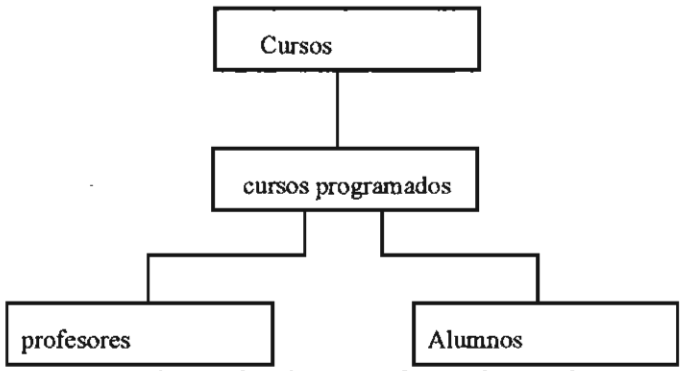


Figura 2. Diagrama de una jerarquía.

En cada nivel de la jerarquía la relación es de uno a muchos. En el diagrama se muestra que un curso puede estar programado varias veces (en diferentes fechas o lugares). Un curso programado puede tener uno o varios alumnos y también puede ser impartido por uno o varios profesores.

La navegación en la jerarquía resulta sencilla si se parte de la raíz del árbol hacia las hojas, como cuando consultamos que alumnos participarán en un determinado curso programado. La dificultad se presenta cuando a partir de las hojas se requiere conocer la información de sus antecesores como al consultar los cursos programados en los que está inscrito un alumno. Para resolver este problema se introduce el concepto de registros virtuales. Estos registros se insertan en la jerarquía pero no contienen información, sino solamente apuntadores a los registros reales con lo que se evita la redundancia. Como ejemplo presentamos una base de datos para representar una lista de materiales ("Bill of materials"), Una estructura física sería como sigue:

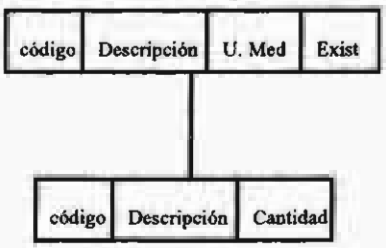


Figura 3. Modelo para una lista de materiales.

Como puede apreciarse para representar que un ensamble está compuesto por cierta cantidad de un determinado componente es necesario introducir cierto nivel de redundancia en el sistema. Esto se evita con la introducción de apuntadores en aquellos lugares en donde podría existir información redundante. Estos apuntadores señalan al registro que contiene la información. A través del subesquema, el usuario percibe esta jerarquía como el árbol del diagrama de la figura 3.

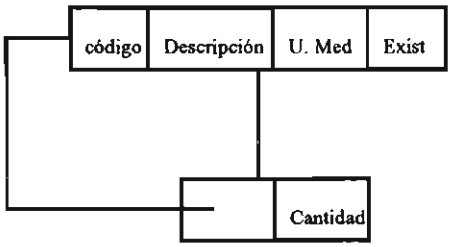


Figura 4. Uso de apuntadores para implementar una jerarquía recursiva.

El modelo jerárquico tiene la ventaja de que si nos permite navegar en la base de datos, sin embargo, cuando necesitamos introducir nuevas relaciones en la base de datos, es necesario realizar una reestructuración mayor. Esto dificultaría la experimentación, y como uno de los objetivos del proyecto es contar con flexibilidad y facilidad para la construcción de los algoritmos de reconocimiento, este modelo no resulta adecuado.

4.3 El modelo de red

El modelo de red fue definido por el DBTG (Database Task Group) de CODASYL (Conference on Data Systems and Languages) [2], en un intento por definir un modelo estándar para el manejo de bases de datos.

El modelo de red también se basa en jerarquías, es decir, relaciones uno a muchos, pero en este caso las jerarquías son de dos niveles únicamente y se denominan sets. Un set se define como una relación de un registro propietario (perteneciente a un archivo propietario) y uno o varios registros miembros (pertenecientes a un archivo miembro).

Para un set se definen las siguientes reglas [2]:

a). Todos sus registros propietarios se toman de un mismo archivo.

b). El registro propietario solamente puede ocurrir una vez como tal en el mismo set.

c). Los registros miembros se toman de un mismo archivo que debe ser diferente del archivo de donde se toman los registros propietarios.

d). Un registro miembro puede aparecer asociado a un solo registro propietario.

e). Todos los registros propietarios o miembros de un set pueden aparecer en otro set como miembros o propietarios siempre y cuando se cumplan las reglas anteriores.

Esquemáticamente un set se representa de la siguiente manera:

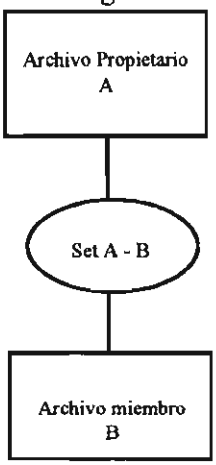


Figura 5. Diagrama de un conjunto (set).

En el modelo de red es posible establecer cualquier número de sets, por lo tanto las jerarquías de más de dos niveles pueden establecerse con varios sets, uno para cada nivel. Las relaciones muchos a muchos entre dos archivos A y B se puede establecer mediante dos sets, en el primero el archivo propietario será A y el archivo miembro el archivo B en tanto que en el segundo set se invierten los papeles. Las relaciones recursivas se establecen con la introducción de registros artificiales. En la figura 6 se muestra como representar la relación jefe-empleados en donde solamente se tiene un archivo de empleados.

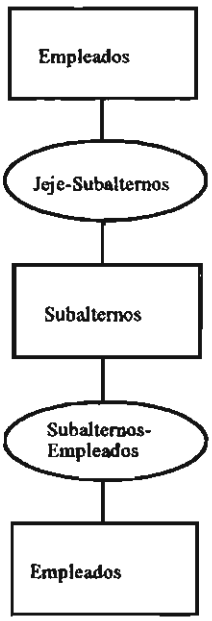


Figura 6. Representación de una relación recursiva en el modelo DBTG.

En esta estructura el set Jefe-Subalternos tiene como propietario al archivo de Empleados y como archivo miembro al archivo Subalternos. El archivo Subalternos es un archivo artificial cuyo único fin es no infringir la regla c. A continuación se define el set Subalternos-Empleados que tiene como archivo propietario al archivo Subalternos y como archivo miembro a Empleados.

Observamos, que respetando las restricciones impuestas por el modelo, es posible construir esquemas de bases de datos que representen correctamente los problemas que se nos presenten.

El modelo de red presenta los mismos problemas respecto a la reorganización de la información en la base de datos, es decir, cada vez que deseamos introducir o eliminar un set, se requiere reorganizar la base de datos. Esto puede ser un verdadero problema para el desarrollo de un sistema experimental en donde de manera continua es necesario probar nuevas estructuras de la base de datos para representar la información visual del robot.

Resumiendo, los tres modelos pueden ser utilizados para la implementación del proyecto, sin embargo, todos presentan inconvenientes de orden práctico. El modelo relacional por su dificultad de navegación y los otros dos por la necesidad de reestructuraciónes al introducir nuevas estructuras de navegación. El sistema propuesto elimina todos esos inconvenientes.

5. Requerimientos del sistema

De acuerdo a [1], una forma conveniente de representación de la información visual del robot es un modelo topológico denominado CW-based. En este modelo se representan los objetos del espacio tridimensional cuantizados en un formato 2 1/2 dimensional, es decir, se representan los objetos como una secuencia de vistas de 2 dimensiones. A manera de ejemplo presentamos el esquema de la representación de un "_Visual_Pattern_Type" (VPT). Un VPT es una estructura para representar un tipo de objetos visual.

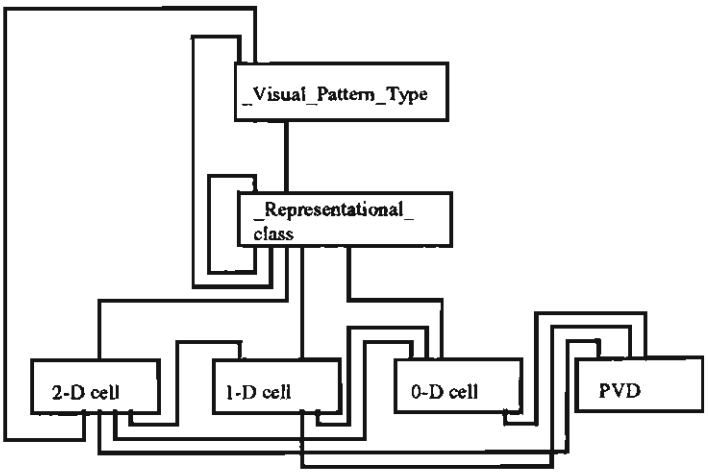


Figura 7. Representación de un patrón visual.

En este diagrama los archivos están representados por rectángulos y los sets por las líneas. En los sets el archivo propietario es el que se conecta con el set por el lado inferior del rectángulo y el archivo miembro es el que se conecta con el set por la parte superior.

Un VPT es propietario de un set que tiene como miembros a varios "_Representational_Classes" (RC's). Un RC representa una vista de un tipo de patrón visual visto desde un punto de vista. El conjunto de RC's miembros de este set representan una cuantización de un objeto tridimensional. Un RC está ligado a sus vecinos visuales los cuales son representaciones del mismo patrón visual que ha sufrido una transformación debido a la aplicación de algún tipo de operador.

Un RC forma sets con sus detalles 2-D (bidimensional), 1-D (unidimensional) y 0-D (cero dimensional). Los detalles 2-D están ligados a sus componentes 1-D y 0-D. Los detalles 1-D forman sets con los detalles 0-D. Los detalles de todos los niveles forman sets con los Detectores Primitivos de Video (PVD's).

El set que tiene como propietario a una celda 2-D y como miembro un VPT representa el conjunto de exclusión, es decir los subcomponentes, que siendo a la vez VPT's no forman parte del VPT original.

Otros objetos de la base de datos tienen estructuras similares por lo que el manejo flexible de los sets es de importancia primaria. Por otro lado, al momento de iniciar el desarrollo no se tienen especificaciones completas de los requerimientos del proyecto de robótica, por esta razón se ha optado por diseñar una estructura que resulte totalmente flexible y que pueda adaptarse fácilmente a nuevos requerimientos que se vayan detectando.

6. Diseño del sistema

Para atender los requerimientos planteados, se adoptó una estructura similar a la de las bases de datos de red con las siguientes ampliaciones:

Un set:

a). Puede establecerse con registros propietarios y miembros tomados del mismo archivo.

b). Un registro específico puede aparecer como propietario y como miembro en diferentes instancias del mismo set.

c). Un registro puede aparecer como miembro dependiente de más de un registro propietario.

d). Un registro propietario solamente lo puede ser, en una instancia del mismo **set**.

Con estas características adicionales, es posible representar las relaciones recursivas con un solo set sin necesidad de recurrir a registros artificiales. En la representación de los vecinos, podemos representar fácilmente el hecho de que un detalle de algún nivel tiene un conjunto de vecinos y a la vez puede ser vecino de varios otros detalles.

En las bases de datos del modelo DBTG, los sets se implementan agregando a los registros de datos los apuntadores necesarios para formar las cadenas . Con esta estrategia, el tiempo de acceso resulta mínimo, pero agregar un nuevo set implica una reorganización de la base de datos. Otro inconveniente son las mismas limitaciones impuestas por el modelo, como el hecho de que un registro miembro solamente puede serlo en una ocurrencia del mismo set y tampoco puede figurar como propietario en el mismo set.

Para evitar estos problemas sin afectar significativamente la eficiencia del sistema se adoptó como estrategia separar los archivos de datos del almacenamiento de las cadenas de los sets. Los archivos de datos no contienen apuntadores, son tablas independientes similares a las utilizadas por las bases de datos relacionales que solamente contienen información del usuario. La única diferencia, es que se agrega un identificador numérico como prefijo del registro en el momento de darlo de alta y que funciona como llave primaria del archivo. Además del identificador, los archivos de datos soportan la definición de índices adicionales con llaves segmentadas. El identificador solamente se agrega en los archivos que va a participar en la estructura de los sets como propietarios o miembros.

La estructura de los sets está representada en un archivo independiente por cada set. En este archivo se graban registros llamados eslabones. Un eslabón representa una relación propietario - miembro. Un eslabón hace referencia a un registro del archivo miembro del set y a un registro del archivo propietario del set. Los eslabones tienen apuntadores con los que

forman dos listas doblemente ligadas entrelazadas. Una lista es la que forman todos los miembros de una instancia de un set y la otra es la de todos los propietarios de un mismo registro miembro en el set. Además, en este archivo se incluyen registros anclas que apuntan al primero y al último eslabón de cada una de las cadenas. Las anclas tienen acceso por llave a fin de poder iniciar el recorrido dentro de la cadena.

Para mantener la consistencia de todos los sets y archivos que participan en ellos, es necesario crear un repositorio que contenga las definiciones de los sets. Para esto se creó una clase de objetos persistentes denominada Database. Este objeto permite controlar la integridad de los sets cuando alguno de los registros de un archivo que participa en los sets es eliminado.

6.1 Los árboles B

Los sistemas de reconocimiento de imágenes utilizan algunos atributos de la imagen para iniciar la búsqueda de patrones similares en la base de datos. Estos atributos pueden ser: la ocurrencia de cierto número de vértices, la presencia de superficies con alguna forma geométrica, etc. Para soportar este tipo de operaciones se requiere de un mecanismo de acceso por llave altamente eficiente y con la flexibilidad de proporcionar acceso por una variedad de conceptos.

Para tener acceso por llave a los registros de los archivos se decidió utilizar los árboles B ya que es un medio sencillo y eficiente que no requiere reorganizaciones de los archivos para mantener un buen tiempo de acceso y un empleo eficiente del espacio en disco.

Los árboles B fueron desarrollados por R. Bayer y E McCreight [7]. En un árbol B los nodos son bloques que pueden contener un número máximo de llaves. Las aristas están representadas por apuntadores a los nodos (bloques) del siguiente nivel. Los apuntadores son simplemente números de bloque. En las hojas los apuntadores a nodos del siguiente nivel son valores nulos.

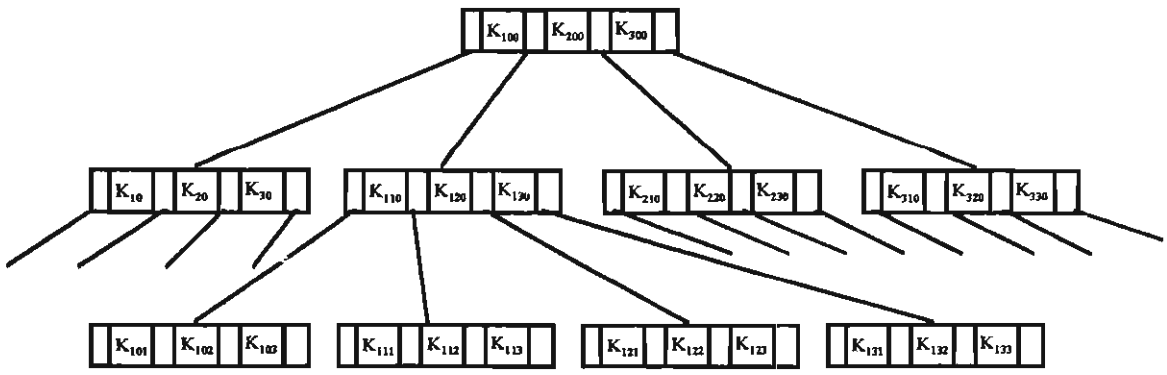


Figura 8. Estructura de un árbol B.

En la figura se muestra parcialmente un árbol B que contiene tres llaves en cada bloque y cuatro apuntadores a los bloques del siguiente nivel. Los algoritmos de inserción y remoción aseguran que todas las hojas del árbol tengan la misma altura, es decir, el árbol permanece balanceado en todo momento. También se asegura que en todos los nodos se mantengan llenos por lo menos a la mitad de su capacidad. En el diagrama los índices de las llaves nos proporcionan una idea del orden de las mismas dentro de la estructura. Dentro de un bloque las llaves se encuentran ordenadas. El subárbol izquierdo de una llave K_j (apuntado por el número de bloque que se encuentra a la izquierda de la llave) contiene llaves menores que K_j en tanto que el subárbol derecho contiene llaves de valor superior a K_j.

El orden *d* de un árbol B es la mitad del número máximo de llaves que puede contener un nodo, entonces, todos los nodos, excepto el nodo raíz contienen por lo menos *d* llaves. La altura de un árbol B con *n* llaves es:[7]

$$h \leq \log_d \frac{n+1}{2}$$

por lo tanto el tiempo de recuperación de una llave es proporcional al logaritmo base *d* de *n*.

Las operaciones de inserción y eliminación en el peor caso recorren el árbol dos veces, una descendiendo de la raíz a una hoja y posteriormente en orden inverso sin embargo el término dominante sigue siendo la altura del árbol y por lo tanto el tiempo de estas operaciones sigue siendo proporcional a $log_d n$.

6.1.1 Algoritmo de inserción

El algoritmo de inserción primeramente localiza el bloque en donde debería encontrarse la nueva llave. Si la llave ya se encuentra en algún nodo, entonces se reporta un error, de otra manera tendremos que llegar hasta una hoja. Si hay espacio en la hoja, la llave se inserta en la posición que le corresponde. En este caso no se varió la altura de ninguna rama.

Si no hay espacio en el bloque entonces se determina si en alguno de los dos bloques más próximos del mismo nivel tienen espacio disponible. Si es así entonces se redistribuyen las llaves de ambos bloques y se cambia la llave en el bloque del que dependen a efecto de mantener la estructura ordenada. En este caso tampoco se altera la altura de ninguna rama.

En el caso de que no existan bloques contiguos con espacio disponible, se crea un nuevo bloque, y todas las llaves, más la que se trata de insertar, se distribuyen en los dos bloques. En este caso una de las llaves tendrá que llevarse al nodo padre, a esta acción se le denomina promoción. La llave promovida se inserta en el nodo padre. Si nuevamente se encuentra lleno, se repite el procedimiento hasta llegar eventualmente al nodo raíz. Si en algún nivel intermedio existe espacio para agregar una llave promovida, entonces no se habrá afectado la

altura de ninguna rama. Si ocurre que el nodo raíz se encuentra lleno, entonces será necesario crear un nuevo nodo raíz, que contendrá una sola llave y dos ramas descendientes. En este caso, la altura de todas las ramas habrá aumentado en uno, manteniéndose así una misma altura para todas ellas.

6.1.2 El algoritmo de eliminación

En este algoritmo, primeramente se localiza la llave que se desea eliminar. Si la llave no se encuentra se reporta un error al usuario.

Posteriormente, se elimina la llave del bloque en que se encuentra. Aquí se presentan dos posibilidades: que la llave se encuentre en una hoja o que el bloque se encuentre en un nodo intermedio.

En cada nodo puede tener como máximo *2d* llaves $K_1, K_2, ..., K_{2d}$ y *2d+1* apuntadores a nodos del siguiente nivel $P_1, P_2, ..., P_{2d+1}$. Si la llave K_i a eliminar se encuentra en un bloque intermedio, para mantener una estructura correcta del árbol B podemos sustituirla por la llave inmediatamente menor que se encuentra en la última posición de la hoja del extremo derecho del subárbol apuntado por P_i. También podemos reemplazarla por la llave inmediata superior que es la primera llave de la primera hoja del subárbol apuntado por P_{i+1}. Después de sustituir la llave por la inmediata mayor o menor, el problema se reduce a eliminar la llave de una hoja.

Al eliminar una llave de una hoja, se puede presentar una condición de deficiencia, es decir, que queden menos de *d* llaves en el bloque. En este caso se pueden redistribuir las llaves con las de algún bloque vecino del mismo nivel y desde luego cambiando la llave de su padre común para mantener la estructura del árbol. Si la operación se puede realizar, no se habrá alterado la altura de ninguna rama, pero si no hay vecinos con más de *d* llaves, entonces será necesario agrupar todas las llaves de los dos bloques en uno sólo y eliminar una entrada en el bloque padre. La llave del bloque padre pasará a formar parte del bloque compactado, el cual queda con *2d* llaves. Si en el padre había más de *d* llaves, termina el proceso sin haberse afectado la altura de ninguna rama. Si este proceso continúa hasta la raíz, y esta queda sin ninguna llave, entonces desaparece y se sustituye por su único descendiente. En este caso la altura de todas las ramas se habrá reducido en uno permaneciendo el árbol balanceado.

6.2 Diseño orientado a objetos

Para el diseño del sistema se adoptó la metodología OMT (Object Modeling Technique) [3] la cual nos proporciona una base sólida para llevar a cabo la programación. En esta metodología se identifican los objetos que van a participar en el sistema, se establecen sus relaciones y se define su funcionalidad. Con los objetos identificados se definieron las siguientes clases:

File. Esta clase contiene los objetos principales para el usuario, o sea, los archivos. Contiene las estructuras de datos y las funciones que le permiten al usuario la manipulación de los archivos como: creación de archivos, grabación de registros, modificación de registros, lecturas y eliminación de registros. Además de las funciones de acceso directo para el usuario, esta clase contiene una serie de funciones de soporte de uso interno.

File_status. Esta clase está contenida dentro de la clase File. Se definió con el fin de separar los datos que representan el estado del archivo para un usuario determinado. Por ahora, el manejador opera en modo monousuario. Se espera que al agrupar los datos de esta clase, facilite en el futuro la conversión a un sistema multiusuario.

Set. Los objetos de esta clase son los sets, es decir, archivos que representan las relaciones uno a muchos entre los registros de un archivo propietario y los registros de un archivo miembro. Dado que esta clase requiere de gran parte de la funcionalidad de la clase File se decidió definirla como su heredera.

File_db. Los objetos de esta clase heredan prácticamente toda la funcionalidad de la clase File. La única diferencia es que estos objetos son los que participan en los sets. Para ello, al crearse se genera automáticamente una llave serial que se usa para identificar los registros dentro de los sets. Otra limitación es la necesidad de utilizar la clase Database para la creación de estos archivos.

Database. Esta clase es una clase contenedora de objetos de la clase File_db y Set. Para crear objetos de estas dos clases, el usuario debe invocar funciones de la clase Database. El objetivo principal de esta clase es conservar las definiciones de los sets de manera permanente. La clase también se responsabiliza de mantener la consistencia de los sets cuando se da de baja algún registro, en este caso, se tienen que revisar todos los sets en los que el registro participa como miembro o propietario para eliminar sus referencias.

Index. Los objetos de esta clase son los índices de los archivos de las clases File, File_db y Set. Sus funciones son de uso interno, es decir, el usuario no las invoca, son invocadas por las funciones de la clase File y sus clases herederas. Mantienen las estructuras de los bloques de índices con funciones de inserción y remoción de llaves y permiten el acceso por llave a los registros de datos. La clase file mantiene una lista de los objetos de la clase Index que representan las llaves del archivo.

Segment. Esta clase no tiene funcionalidad propia, pero contiene la información que describe los segmentos de las llaves. Los objetos de la clase Index mantienen una lista de objetos de la clase Segment que representan los segmentos de la llave.

Cache. La clase cache es la responsable de manejar las interfaces con el sistema operativo en cuanto a la entrada y salida. La entrada y salida se maneja a nivel de bloques de tamaño fijo. La clase cache mantiene un conjunto de buffers y mediante una lista de los más recientemente utilizados, minimiza los accesos a disco.

Aligner. Esta es una clase de utilería que se usa para alinear los segmentos de una llave de acuerdo al equipo en donde se usa el sistema. Aligner identifica los requerimientos de alineación de cada tipo de dato y construye una tabla que utiliza para realizar las alineaciones.

El diseño general del sistema se muestra en la figura 9.

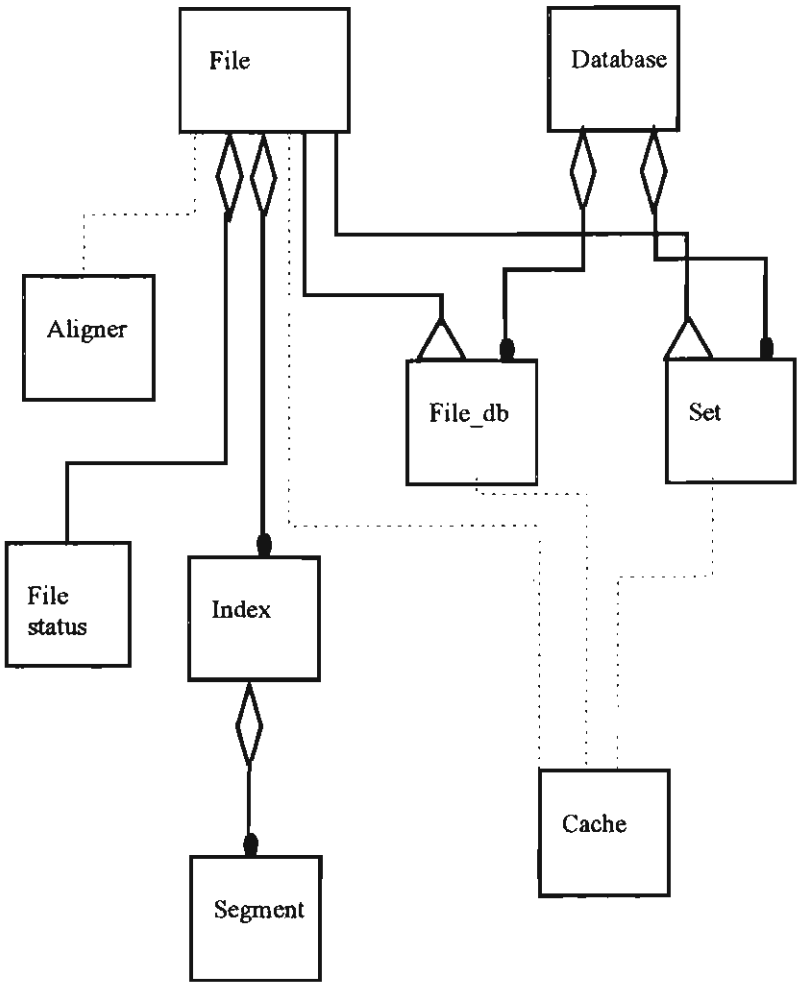


Figura 9. Clases de clases del sistema UAMRM.

En la notación de OMT se representan las clases como rectángulos y las relaciones entre clases como líneas con diferentes terminadores. Cada terminador representa un tipo de relación. En la figura 9 podemos observar las siguientes relaciones: la línea con el rombo

indica pertenencia o composición, si termina en un círculo negro la relación es de uno a muchos. La línea que termina en un triángulo representa la relación de herencia. Y finalmente hemos representado la relación de uso con una línea punteada. De esta manera tenemos:

1). La clase File tiene dos subclases. La clase File_db y la clase Set. La primera representa los archivos de la base de datos, mientras que la segunda se usa para representar los Sets. A su vez, la clase File no es una clase abstracta ya que podemos definir archivos que no necesariamente pertenezcan a la base de datos. La funcionalidad adicional que tiene la clase File_db sobre la clase FILE es que cuando se crea un objeto de esta clase, se le define un índice para identificar los registros dentro de los sets. En la clase Set además se cuenta con los mecanismos para manipular las cadenas con que se representan los Sets.

2). La clase Database está compuesta por las clases File_db y Set. En este caso Database lleva el control de la composición de los sets y mantiene la integridad de los mismos.

3). La clase File está compuesta por una ocurrencia de la clase File_status la cual contiene la información relativa al estado del archivo para un usuario.

4). La clase File, también contiene uno o varios objetos de la clase Index, uno por cada uno de los índices que contiene el archivo. Cada objeto de la clase Index contiene uno o varios objetos de la clase Segment los cuales describen los segmentos que constituyen las llave segmentadas.

5). La clase Cache es utilizada por las clases File, File_db, y Set. La clase Cache es la responsable de mantener el cache de buffers para la entrada y salida. En el momento de su creación se asignan automáticamente un cierto número de buffers. El usuario no tiene que preocuparse por la inicialización de este objeto.

6). La clase Aligner se definió porque proporciona el servicio de alineación de los segmentos de llaves. Estas funciones requieren de la inicialización de una tabla que depende de la arquitectura del equipo en donde opera el sistema. Al definirse como objeto se aprovechó su método constructor que es el que se encarga de inicializar la tabla de alineación.

6.3 Implementación del sistema

Para el desarrollo del sistema se empleó el compilador Microsoft Visual C++ Ver. 1.52, sin embargo, se cuido de utilizar únicamente las opciones ANSI a fin de poder transportar el sistema a otras plataformas.

6.3.1 Estructura de los archivos

Los archivos de datos están contenidos en bloques de tamaño fijo. Los registros de datos y los bloques de índices residen en el mismo archivo. Además, el primer bloque del archivo contiene el descriptor del archivo. El resto de los bloques pueden ser: Bloques de datos, bloques de índices o bloques libres (los que han sido liberados después de haberse utilizado). Los bloques de índices son manejados por la clase Index.

6.3.1.1 El bloque descriptor

El descriptor proporciona información acerca de los registros de datos utilizados, los bloques libres disponibles y la estructura de los índices. El bloque descriptor contiene los siguientes datos:

filesize. El tamaño total del archivo en número de bloques, incluye el bloque descriptor, los bloques de datos, los bloques de índices y los bloques libres. Este dato se utiliza para obtener el siguiente número de bloque cuando crece el archivo.

fstdatablk. Primer bloque de datos del archivo (respecto al orden de inserción). Este dato nos permite posicionar el archivo en el primer bloque de datos para leerlo en orden de su secuencia física..

lstdatablk. Último bloque de datos del archivo. Con esta información es posible posicionar el archivo en el último bloque de datos para leerlo en orden inverso respecto al orden de inserción.

reccnt. Número total de registros lógicos en el archivo.

serial. Un número secuencial que se incrementa en uno cada vez que se asigna un valor a un segmento de llave tipo serial.

freeblk. Los bloques de un archivo que se liberan durante las operaciones de remoción se agregan a una lista para usarlos cuando se requiera un nuevo bloque. Este dato apunta al primer bloque de la lista de bloques libres.

idxnr. Número de índices definidos en el archivo.

idx. Vector de descriptores de índices. Cada elemento contiene la siguiente información.

name. Nombre del índice. Con este nombre se hace referencia al índice que se va a usar para recuperar la información.

segnr. Número de segmentos que constituyen la llave. Cada segmento es un dato elemental.

fstseg. El número del primer descriptor de segmentos usado para la llave.

rootblk. Número del bloque raíz de la estructura de índices.

keylen. Longitud total de la llave. Es la suma de las longitudes de los segmentos.

keys_in_blk. El número máximo de llaves que puede contener un bloque de índices.

seg. Vector que contiene los descriptores de segmentos. Cada elemento de este vector es una estructura idx_seg la cual contiene los siguientes datos:

offset. Desplazamiento en bytes del segmento dentro del registro de datos.

len. Longitud del segmento.

type. Tipo de dato del segmento. El dato se encuentra codificado con la siguiente tabla:

SEG_TYPE_CHAR	0	char
SEG_TYPE_INT	1	int
SEG_TYPE_UINT	2	int
SEG_TYPE_LONG	3	long
SEG_TYPE_ULONG	4	unsigned long
SEG_TYPE_FLOAT	5	float
SEG_TYPE_DOUBLE	6	double
SEG_TYPE_SERIAL	7	Serial

desc. Valor que identifica si el valor del segmento se toma en orden ascendente o descendente (0. ascendente, 1. descendente)

La figura 10 muestra esquemáticamente la forma como se representan las llaves de un archivo en el bloque descriptor.

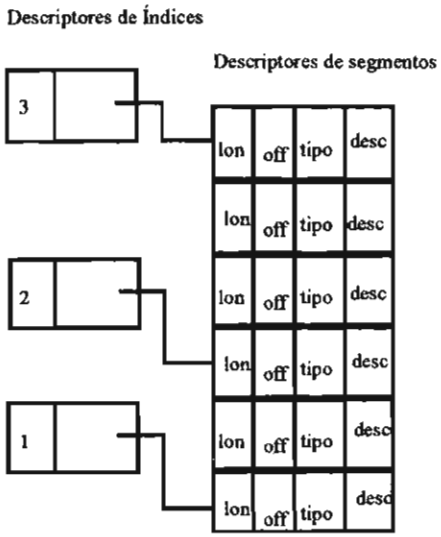


Figura 10.- Descriptores de los segmentos de llaves de un archivo.

La figura 10 muestra un archivo con tres índices. Las llaves contienen tres, dos y un segmento respectivamente.

El bloque descriptor se graba en el disco en el momento de cerrarse el archivo. Cuando se abre un archivo que ya existe, la información del bloque se utiliza para inicializar los objetos de la clase File.

6.3.1.2 Los bloques de datos

Los bloques de datos se encuentran formando una lista doblemente ligada para poder realizar recorridos hacia adelante y hacia atrás. En el bloque descriptor se encuentran los números de bloque del primero y último de los bloques de datos. La figura 11 muestra la estructura de la cadena de bloques de datos.:

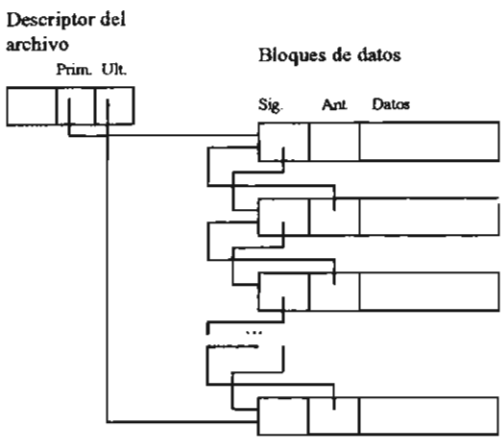


Figura 11. Cadenas doblemente ligadas de los bloques de datos.

Los registros del bloque de datos son de longitud variable, limitados a la capacidad del bloque. Los bloques de datos contienen la siguiente información:

prevblk. Número de bloque de datos anterior.

nextblk. Número del siguiente bloque de datos.

used. Número total de espacio usado en el bloque. Incluye estos datos.

records. Los registros de datos del usuario.

Los registros de datos se graban precedidos de su longitud y se encuentran siempre en forma contigua. Cuando se elimina un registro el resto de los registros se recorren hacia el principio del bloque.

6.3.1.3 Los bloques de índices

Los bloques de índices son utilizados para construir las estructuras de los árboles B. Contienen la siguiente información:

num_keys. El número de llaves que contiene el bloque

ib_index_blk[]. Este vector contiene los números de bloque del siguiente nivel del árbol B.

ib_data_blk[]. Contiene los números de bloque en donde se encuentran los registros de datos asociados a las llaves.

ib_key[]. Es el vector de llaves contenidas en el bloque.

ib_index_blk, ib_data_blk e ib_key son dinámicos por esto no están definidos en la estructura del bloque. Estos vectores están definidos como macros en el archivo file.h. Las macros hacen referencia a los siguientes parámetros que son miembros de la clase Index:

index_block_off. Desplazamiento del vector ib_index_blk en el bloque de índices.

data_blk_off. Desplazamiento del vector ib_data_blk en el bloque de índices.

key_off. Desplazamiento del vector de llaves en el bloque de índices.

Estos tres valores se calculan en la función constructora de la clase Index, la cual se invoca cuando se crea el índice y cuando se abre el archivo.

6.3.2 Estructura de los sets

Cada set se maneja como un archivo independiente. En este archivo se mantienen dos tipos de registros: las anclas y los eslabones. Por cada ocurrencia de un set existe una ancla identificada por su registro propietario que apunta al primer eslabón de la cadena de los registros miembros del **set**. Las anclas tienen como llave el identificador del registro propietario y se mantiene un índice de árbol B para su rápida localización.

Se tomó la decisión de crear un mecanismo similar para los registros miembros, es decir, por cada registro que aparece como miembro del **set** se tienen una ancla que apunta a la cadena de todos los registros que aparecen como propietarios de ese registro. Con esto se pueden tener búsquedas de todos los propietarios de un miembro. Para la cadena de los registros propietarios se utilizan los mismos eslabones que se usaron para crear las cadenas de los miembros.

Un registro eslabón establece la relación de pertenencia en una ocurrencia de un **set** entre el registro propietario y el registro miembro. Los registros eslabones mantienen dos listas doblemente ligadas enlazadas, es decir, un eslabón contiene apuntadores al siguiente y al anterior registro miembro de un set y otro par de apuntadores para localizar al siguiente y al anterior propietario del mismo registro miembro.

El registro ancla contiene los siguientes datos:

type. Identifica el tipo de registro de la siguiente manera:
REC_TYPE_ANCHOR_PROP(1).- Registro ancla de registro propietario
REC_TYPE_ANCHOR_MEMB(3).- Registro ancla de registro miembro

serial. Identificador serial del registro propietario o miembro, para registros con tipo 1 ó 3 respectivamente.

first_link. Apuntador al primer eslabón de la cadena de registros miembros o propietarios para los tipos de registro 1 ó 3 respectivamente.

last_link. Apuntador al último eslabón de la cadena de registros miembros o propietarios para los tipos de registro 1 ó 3 respectivamente.

Los registros eslabón que forman las cadenas de los sets contienen los siguientes datos:

type. Identificador que indica que es un eslabón. REC_TYPE_LINK (2).

prefix. Apuntador a él mismo.

owner_record. Identificador serial del registro propietario.

member_record. Identificador serial del registro miembro.

next_link_owner. Apuntador al siguiente eslabón de la cadena del registro propietario.

prev_link_owner. Apuntador al eslabón anterior de la cadena del registro propietario.

next_link_memb. Apuntador al siguiente eslabón de la cadena del registro miembro.

prev_link_memb. Apuntador al eslabón anterior de la cadena del registro miembro.

En la figura 12 se muestra un set que contiene dos ocurrencias. La primera tiene como propietario un registro A y como miembros los registros P y Q. La segunda instancia tiene como propietario al registro B y como único miembro al registro P. Es fácil comprobar como a partir del ancla del registro propietario A podemos localizar todos los registros miembros del set. También se puede apreciar como a partir del ancla miembro P podemos localizar todo los registros propietarios asociados a B en alguna instancia del set.

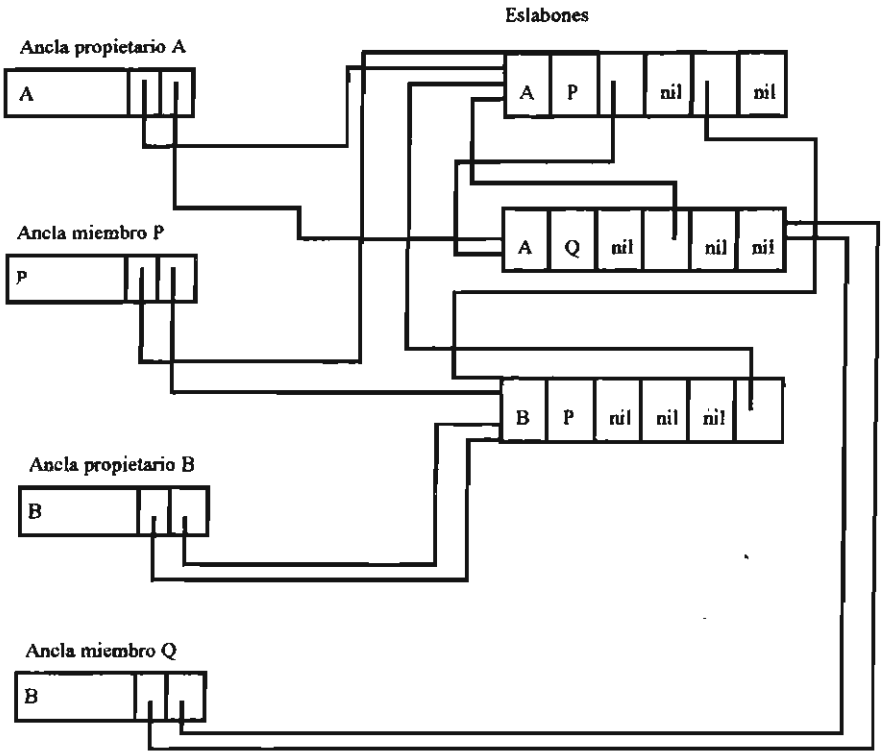


Figura 12. Representación de los "sets".

6.3.3 Estructura de la clase Database

En este documento aplicamos el término base de datos a un conjunto de archivos de la clase File_db y un conjunto de sets que se encuentran relacionados. La clase **Database** define un objeto persistente que se encarga de registrar la existencia de todos los archivos de la clase File_db y sets que constituyen la base de datos.

Un objeto de la clase Database, es el responsable de registrar los archivos que participan en los sets y los sets mismos. Esta información se mantienen dentro del objeto y está disponible para posteriores usos del sistema.

Para lograr que el objeto Database pueda controlar de manera efectiva la base de datos, toda solicitud de los usuarios para la creación o destrucción de archivos de datos File_db y sets debe ser atendida por este objeto.

7. Diseño funcional del sistema

Para el diseño de las funciones miembro de cada una de las clases que participan en el sistema, se adoptó la metodología de diseño estructurado de Edward Yourdon[5], la cual puede combinarse con la metodología orientada a objetos, al definir las funciones miembro de cada una de las clases. Uno de los elementos más importantes de la metodología es el diagrama de estructura. Un diagrama de estructura es un diagrama jerárquico que muestra los módulos que constituyen un sistema y como estos módulos se encuentran conectados. En el diagrama, el módulo principal se muestra en la parte superior y está conectado con líneas con los módulos del siguiente nivel. La línea representa una relación de modulo llamador a módulo llamado.

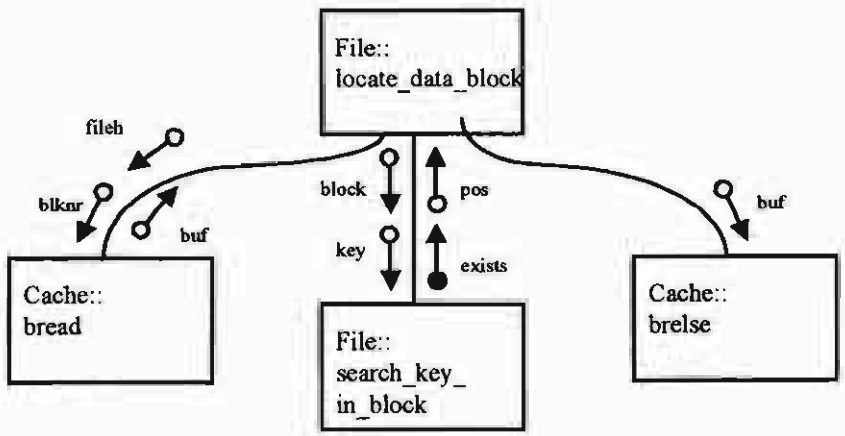


Figura 13. Un diagra de estructura.

La figura 13 muestra una estructura simple en la que el módulo File::locate_data_block invoca a las funciones Cache::bread, File::search_key_in_block y Cache::brelse. En el diagrama aparecen pequeñas flechas etiquetadas que indican el intercambio de parámetros y valores entre la función llamadora y la función invocada. Una flecha en dirección al módulo llamado, indica un parámetro para el módulo. Una flecha en dirección contraria, indica un valor regresado por el módulo invocado. El círculo vacío en el extremo de la flecha, indica un dato, en tanto que el círculo lleno indica información de control. En el diagrama, al invocarse al módulo Cache::bread se le pasan como argumentos fileh y blknr que son el asa de un archivo y el número de bloque que se desea respectivamente. La función regresa buf que es un apuntador al buffer que contiene el bloque solicitado. En el llamado a File::seach_key_in_block, el módulo regresa el dato pos que es la posición de la llave en el bloque y un indicador para saber si la llave se encontró en el bloque.

7.1 Diseño funcional de la clase File

La clase File es la responsable del manejo de los archivos. Su funcionalidad consiste de las operaciones básicas de entrada y salida. Esta funcionalidad se comparte con sus clases herederas como son File_db y Set. Contiene los siguiente elementos de información:

fileh. Asa que proporciona el sistema operativo al abrir o crear el archivo. Identifica al archivo en todas las operaciones de entrada y salida.

status. Indicador que tiene dos valores:
FILE_STATUS_CREATED. El archivo se ha creado.
FILE_STATUS_OPEN. El archivo ya existía y fue abierto.

db. Apuntador al objeto Database que controla el archivo. Esta es una referencia circular que se utiliza para invocar funciones miembro de la clase Database..

filesize. El tamaño total del archivo en número de bloques, incluye el bloque descriptor, los bloques de datos, los bloques de índices y los bloques libres. Este dato se utiliza para obtener el siguiente número de bloque cuando crece el archivo.

fstdatablk. Primer bloque de datos del archivo (respecto al orden de inserción). Este dato nos permite posicionar el archivo en el primer bloque de datos para leerlo en orden de su secuencia física.

lstdatablk. Último bloque de datos del archivo. Con esta información es posible posicionar el archivo en el último bloque de datos para leerlo en orden inverso respecto al orden de inserción.

reccnt. Número total de registros lógicos en el archivo.

serial. Un número secuencial que se incrementa en uno cada vez que se asigna un valor a un segmento de llave tipo serial.

freeblk. Los bloques de un archivo que se liberan durante las operaciones de remoción se agregan a una lista para usarlos cuando se requiera un nuevo bloque. Este dato apunta al primer bloque de la lista de bloques libres.

idxnr. Número de índices definidos en el archivo.

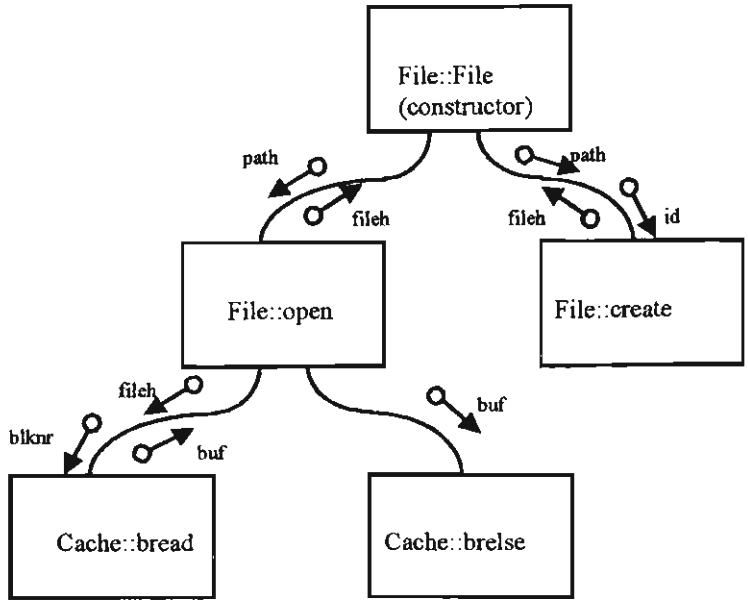
first_index. Apuntador al primer objeto Index de la cadena de descriptores de índices.

file_status. Una estructura que refleja el estado particular del archivo para un usuario.

7.1.1 File::File (Constructora)

La Función constructora detecta la existencia del archivo. Si el archivo ya existe lo abre y lee sus parámetros del bloque descriptor. Si no existe lo crea e inicializa sus parámetros.

Diagrama de estructura:



Precondiciones:

No hay.

Poscondiciones:

El archivo se encuentra abierto.

Entradas:

Path	Ruta del archivo en el sistema de archivos.
id	Identificador del archivo en la base de datos.

Proceso:

Si el archivo existe
- Abre el archivo
  - Lee el bloque descriptor del archivo
  - Almacena la información del descriptor en las variables del objeto
  - Inicializa el estado del archivo
  - Libera el bloque descriptor del archivo
  - Crea las estructuras de índices y descriptores de segmentos
- Marca el archivo como abierto

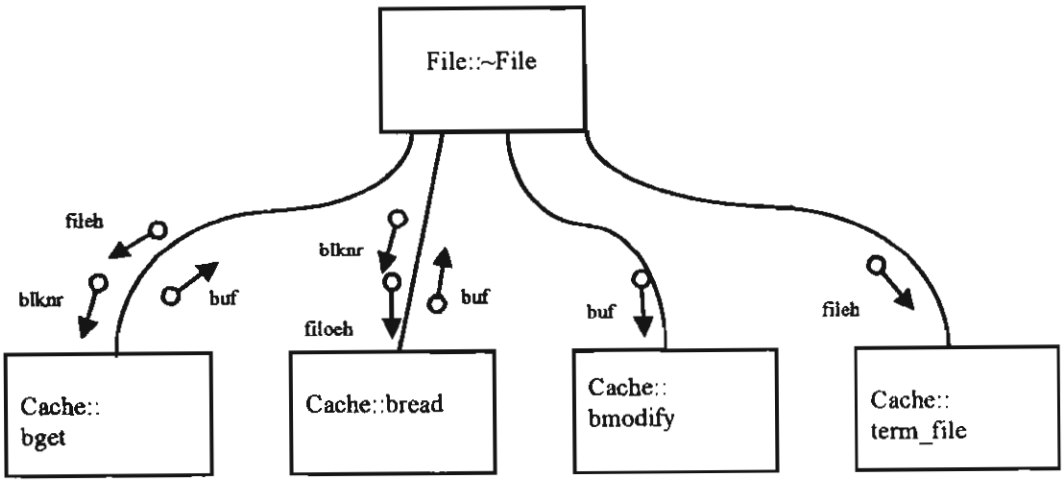
Sino
- Crea el archivo

Abre un nuevo archivo en el sistema
Inicializa las variables del objeto
Inicializa el estado del archivo
Marca el archivo como creado

7.1.2 File::~File (Destructora)

La función destructora restaura en el bloque descriptor toda la información relativa al estado del archivo. En esta operación se registran los nuevos índices que se hayan creado en el archivo.

Diagrama de estructura:



Precondiciones:

El archivo fue abierto o creado (mantienen un indicador de esta condición).

Poscondiciones:

El archivo se cierra y toda su información queda permanente en disco.

Entradas:

No hay.

Salidas:

No hay.

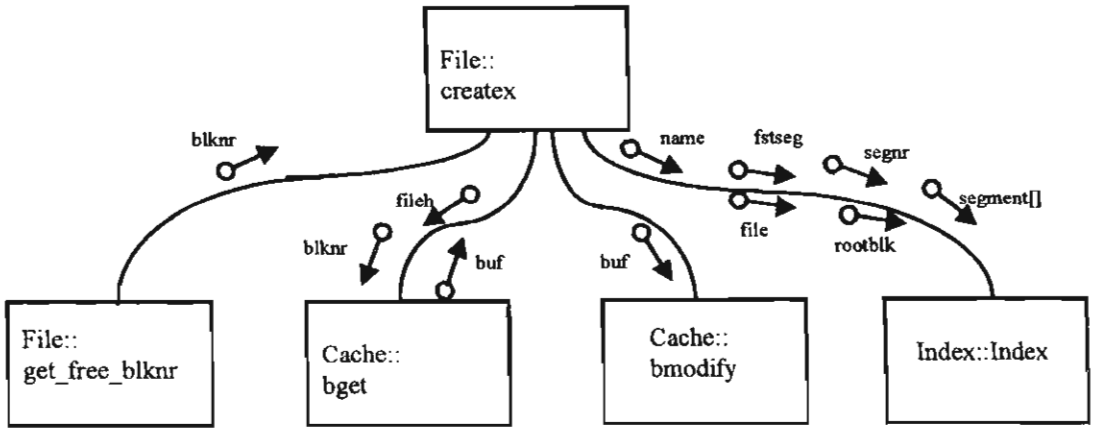
Proceso:

Si el archivo fue creado durante la sesión de trabajo
    Se solicita un nuevo bloque para el descriptor del archivo
sino
    Se lee el bloque descriptor del archivo
Actualiza los parámetros del archivo
Por cada uno de los índices del archivo
    Registra el descriptor del índice en el bloque descriptor del archivo
    Por cada uno de los segmentos del índice
        Guarda el descriptor del segmento en el bloque descriptor del archivo
Marca como modificado el bloque descriptor del archivo
Vacía a disco todos los bloques del archivo que se encuentran en el cache

7.1.3 File::createx (Crea un nuevo índice)

Esta función crea una nueva estructura de índices de árbol B para el archivo. El índice se crea con su bloque raíz vacío. Si existen registros de datos en el archivo se incorporan todas las llaves al índice.

Diagrama de estructura:



Precondiciones:

El archivo debe estar abierto.
Debe haber una entrada disponible en el vector de índices.
Debe haber espacio suficiente para almacenar los descriptores de segmentos.
Que no exista un índice con el mismo nombre.

Poscondiciones:

El índice tiene un bloque raíz vacío.

Entradas:

name — Cadena de hasta 9 caracteres que se usa como identificador del índice. Este nombre será usado en las referencias futuras al índice.

segnr — Número de segmentos que constituyen la llave.

seg — Apuntador al primer elemento de un vector que contiene los descriptores de los segmentos de la llave. Cada elemento del vector es una estructura como sigue:

```
struct idx_seg {
        int    offset;          // offset in record
        int    len;          // Segment Length
        char   type;          // SEG_TYPE_XXX
        char   desc;          // Descending Order
};
```

en donde:

offset — Es el desplazamiento del segmento dentro del registro de datos (cero para el primer dato).

len — Longitud en bytes del segmento.

type — Tipo de segmento (puede usarse la siguiente tabla de constantes)

SEG_TYPE_CHAR	char
SEG_TYPE_INT	int
SEG_TYPE_UINT	unsigned int
SEG_TYPE_LONG	long
SEG_TYPE_ULONG	unsigned long
SEG_TYPE_FLOAT	float
SEG_TYPE_DOUBLE	double
SEG_TYPE_SERIAL	Serial

desc — Indica si el ordenamiento (0. Ascendente, 1. Descendente)

Salidas:

No hay.

Proceso:

Verifica las precondiciones
Localiza el final de la cadena de descriptores de índices
Calcula fstseg como la suma del número de segmentos de todos los índices
Solicita un número de bloque para el bloque raíz
Solicita un buffer para el bloque raíz
Inicializa el bloque con cero llaves
Marca el bloque como modificado
Crea un nuevo índice
Incrementa el número de índices del archivo
Por cada registro de datos en el archivo
    Genera la llave que le corresponde
    Incorpora la llave generada en la estructura del índice

7.1.4 File::deletex (Elimina un índice)

Elimina un índice del archivo.

Diagrama de estructura:

n/a (no aplicable)

Precondiciones:

El índice debe existir en el archivo.

Poscondiciones:

El índice se elimina del archivo.
Los bloque liberados quedan disponibles para el futuro crecimiento del archivo.

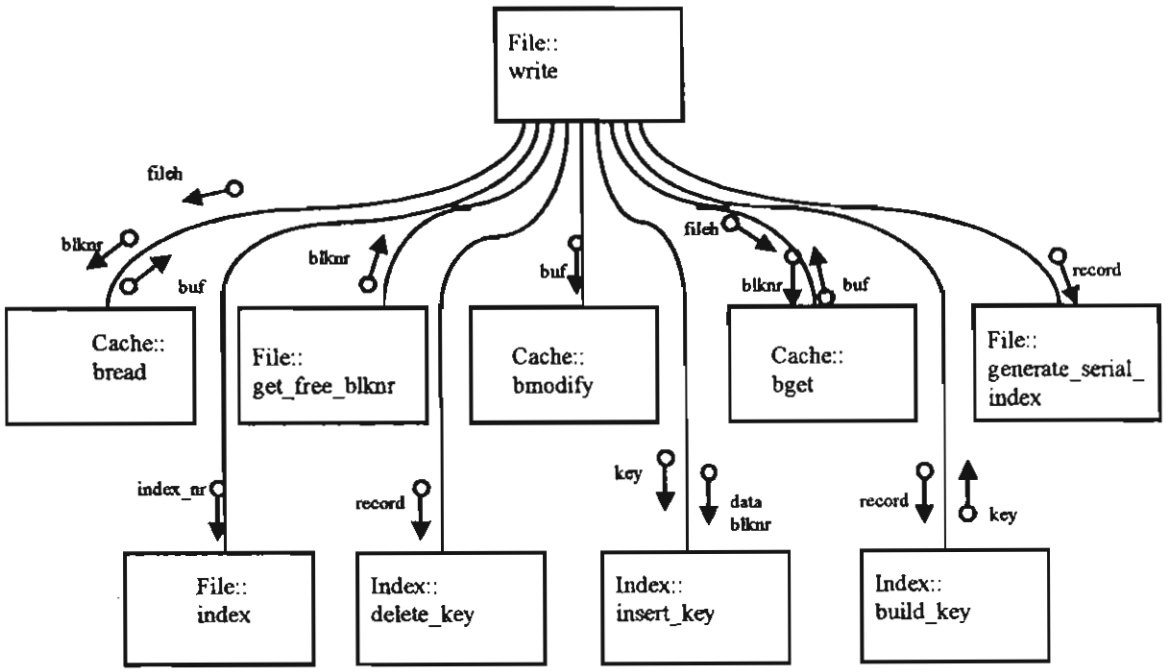
Proceso:

Localiza el índice en la lista de índices del archivo
Si se encuentra
    Elimina la estructura del índice
    Elimina la referencia del índice de la lista de índices
sino
    Reporta un error y da por terminado el programa

7.1.5 File::write (Escribe un nuevo registro)

Esta función agrega un nuevo registro al archivo de datos. Genera todas las llaves y las inserta en las estructuras de índices definidas para el archivo.

Diagrama de estructura:



Precondiciones:

No deben existir llaves duplicadas.

Poscondiciones:

El registro se incorpora en un bloque de datos.
Se crea una entrada para cada índice del archivo.

Entradas:

record	Buffer que contiene el registro a incorporar.
recsize	Longitud en bytes del registro.

Proceso:

2893176

Si no hay bloques de datos (es el primer registro de datos)
Crea un nuevo bloque
Inicializa la lista de bloques de datos

```
        Inicializa el espacio del bloque
sino
        Lee el último bloque de datos
Si no hay espacio suficiente para el registro
        Solicita un número de bloque
        Encadena el último bloque de datos con el nuevo número
        Marca el bloque lleno como modificado
        Solicita un buffer para el nuevo bloque
        Inicializa el nuevo bloque
Genera las llaves seriales en el área del registro
Salva en índice actual
Por cada uno de los índices del archivo
        Establécelo como índice actual
        Genera la llave con la información del registro
        Incorpora la llave al índice
        Si la llave resultó duplicada
                Por cada uno de los índices ya modificados
                        Elimina la llave del índice
                Regresa condición de llave duplicada
Restablece el índice actual
Mueve el registro de datos al bloque
Actualiza el espacio utilizado en el bloque
Marca el bloque como modificado.
```

7.1.6 File::generate_serial_index (Genera los segmentos seriales)

Esta función se invoca automáticamente cuando se dá de alta un registro. Su propósito es generar, en el registro de datos, los segmentos de llave de tipo serial que aparecen en cualquiera de las llaves.

Precondiciones:

No hay.

Poscondiciones:

Todos los segmentos de llaves de tipo serial contienen el siguiente número serial asignado por el sistema en el registro de datos.

Entradas:

rec	Registro de datos que se va a incorporar en el archivo

Salidas:

No hay.

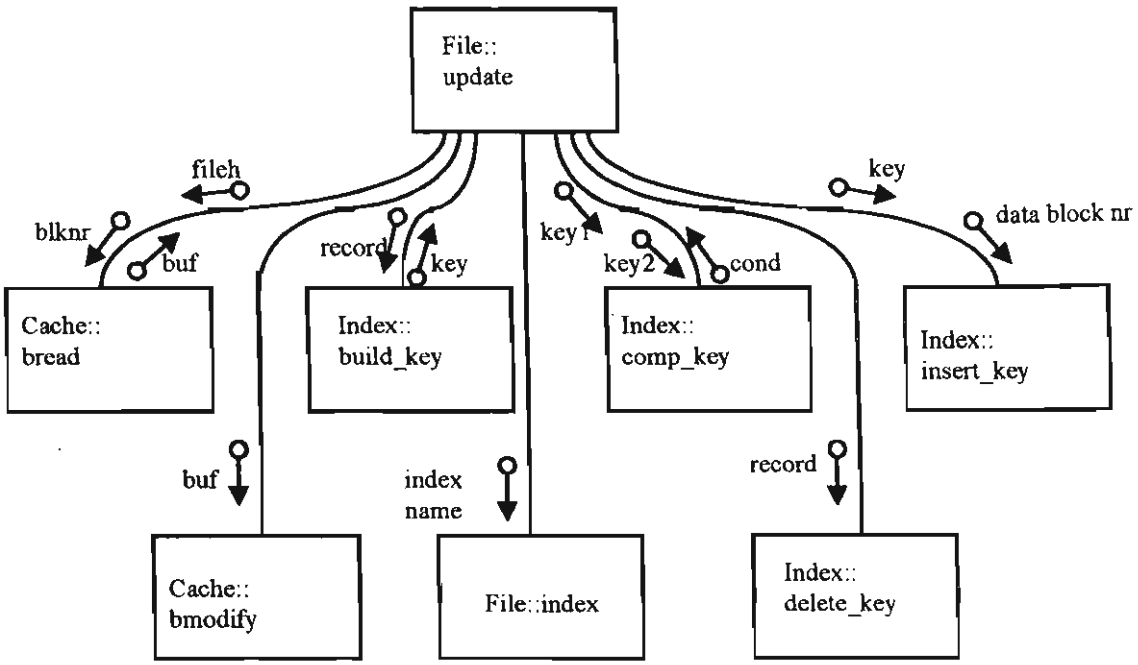
Proceso:

Por cada índice del archivo
Por cada segmento del índice
Si el segmento es de tipo serial
Mueve al registro de datos el siguiente número serial

7.1.7 File::update (Modifica el registro actual)

El registro leído o grabado más recientemente es actualizado en su sitio. No se puede modificar su longitud ni sus segmentos de llave de tipo Serial.

Diagrama de estructura:



Precondiciones:

El registro a modificar debe haber sido el último registro leído en el archivo.

Poscondiciones:

El registro contiene la nueva información.

Se actualizan todas las llaves que hayan cambiado.

Entradas:

record Registro de datos con el nuevo contenido.

Salidas:

No hay.

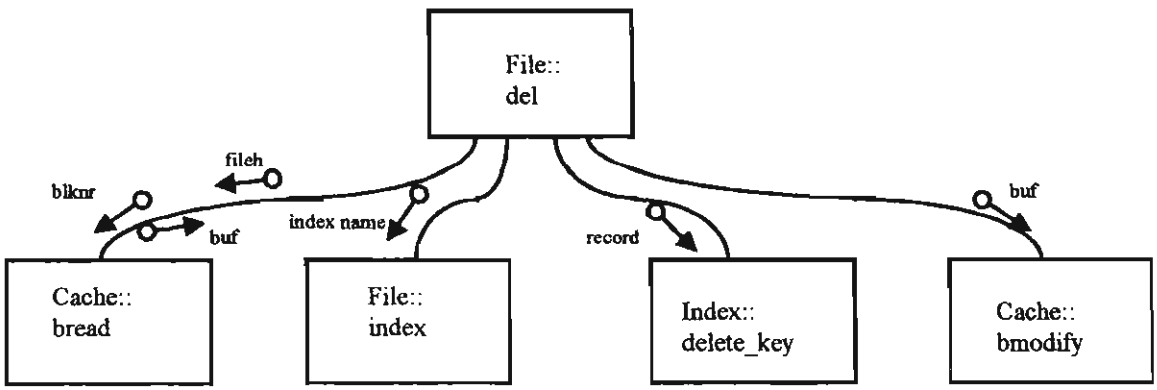
Proceso:

Lee el bloque de datos actual
Determina la posición del registro actual dentro del bloque
Salva el índice actual
Por cada uno de los índices del archivo
    Establece el índice como índice actual
    Genera la llave new_key con la información del nuevo registro
    Genera la llave old_key con la información contenida en el bloque de datos
    Si las llaves son diferentes
        Elimina old_key del índice
        Incorpora new_key en el índice
reestablece el índice actual
Mueve el registro al bloque de datos
Marca el bloque de datos como modificado

7.1.8 File::del (Elimina el registro actual)

La función del elimina un registro de datos del archivo. Además elimina las entradas correspondientes a las llaves que le corresponden en las estructuras de índices definidas para el archivo.

Diagrama de estructura:



Precondiciones:

El registro a eliminar es el registro actual.

Poscondiciones:

El registro se elimina físicamente del archivo.
Se eliminan la llave correspondiente a cada índice

Proceso:

Lee el bloque de datos actual
Localiza el registro en el bloque de datos
Salva en índice actual
Por cada uno de los índices
    Establece el índice como actual
    Elimina la llave que le corresponde al registro
Restaura el índice actual
Elimina físicamente el registro del bloque de datos
Marca el bloque de datos como modificado

7.1.9 File::index (Establece el índice actual)

Esta función establece uno de los índices del archivo como índice actual:

Precondiciones:

No hay.

Poscondiciones:

El índice seleccionado permanece como índice actual.

Entradas:

name	Nombre del índice que se selecciona como índice actual.

Salidas:

Index	Apuntador al objeto Index que queda como índice actual 0 si no se localizó el índice.

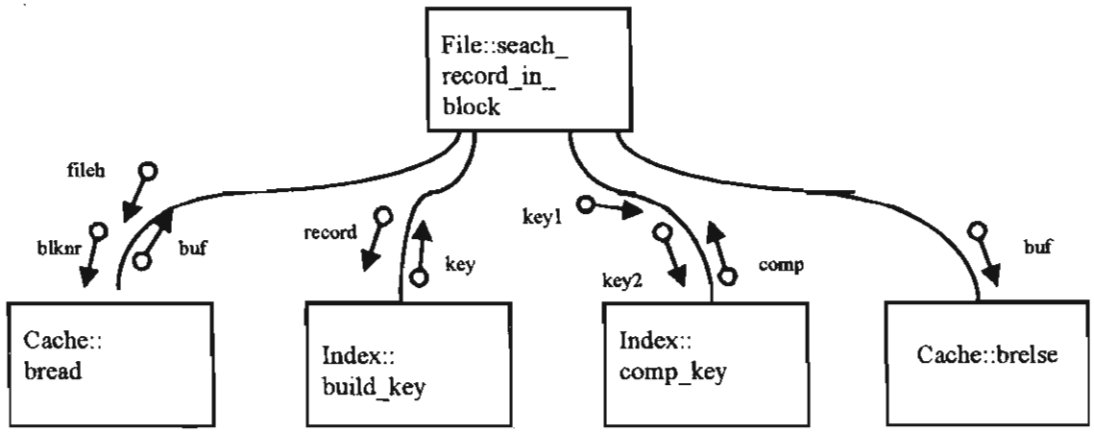
Proceso:

Por cada índice del archivo
Si el nombre del índice es igual al parámetro
Establece el índice como índice actual
Regresa el apuntador al índice
Regresa 0

7.1.10 File::seach_record_in_block (Localiza un registro en un bloque de datos)

Con una llave que corresponde al índice actual se localiza el registro correspondiente en un bloque de datos.

Diagrama de estructura:



Precondiciones:

El índice actual debe corresponder a la llave que se proporciona como argumento.

Poscondiciones:

No hay.

Entradas:

blknr	Número del bloque de datos en el que buscará el registro
key_arg	Llave de búsqueda
rec	Área en donde se deja el registro localizado

Proceso:

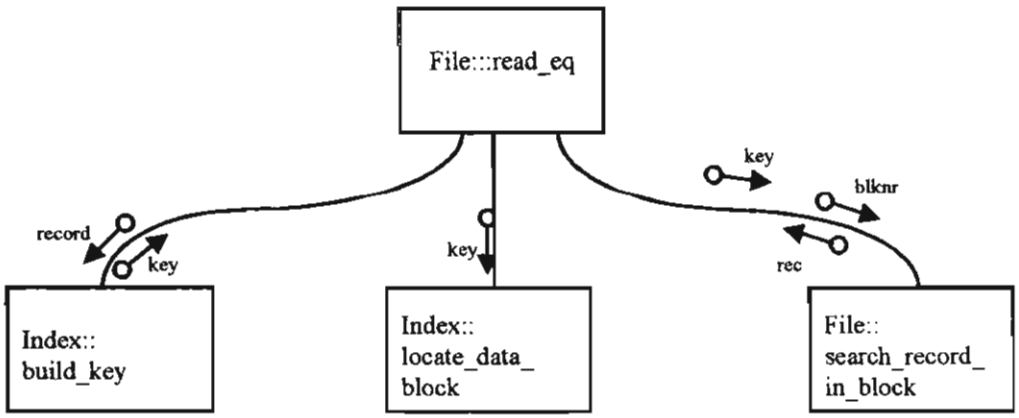
Lee el bloque de datos

Posiciona el bloque en el primer registro de datos
Mientras haya más registros de datos
    extrae la llave del bloque en key_blk
    Si la key_blk es igual a key_arg
        Mueve el registro del bloque a rec
        Si el primer segmento de la llave es tipo serial
            Mueve el dato serial al serial actual
        Regresa la longitud del registro
    Avanza al siguiente registro del bloque
Error, el registro no se localizó en el bloque

7.1.11 File::read_eq (Lectura random por llave igual)

La función File::read_eq lee el registro cuya llave corresponde a la llave contenida en el registro que se proporciona como argumento. La lectura se realiza mediante el índice actual.

Diagrama de estructura:



Precondiciones:

- La llave actual debe corresponder a la llave que se proporciona como argumento.

Poscondiciones:

El registro leído se establece como registro actual

Entradas:

rec	Apuntador al área que contiene la llave y en donde se dejará el registro leído.

Salidas:

long	Longitud en bytes del registro leído (Si hubo éxito) cero si no se encontró el registro.

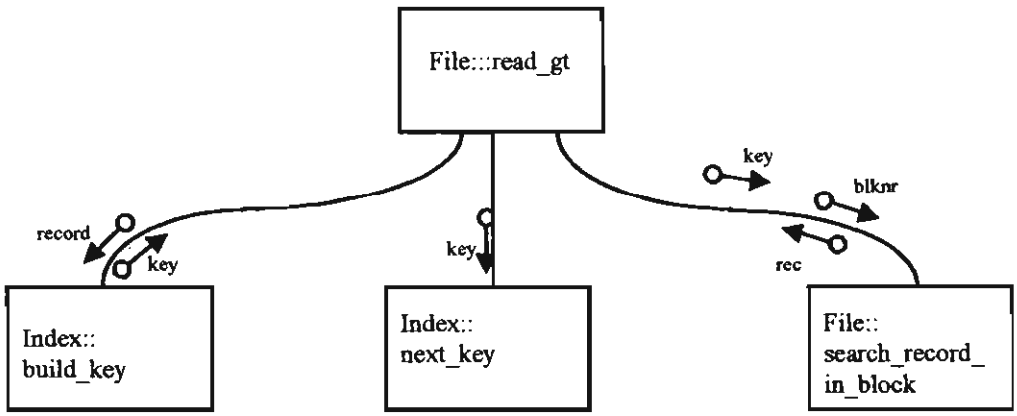
Proceso:

Extrae la llave del registro
Localiza el bloque de datos en donde se encuentra el registro
Si no se encontró
    Regresa 0
Localiza el registro dentro del bloque de datos

7.1.12 File::read_gt (Lee un registro con una llave mayor)

Lee el registro con la menor llave que es mayor de la llave contenida en el registro que se proporciona como argumento.

Diagrama de estructura:



Precondiciones:

La llave actual debe corresponder a la llave de referencia para la búsqueda.

Poscondiciones:

El registro leído se establece como registro actual.

Entradas:

rec Apuntador al área que contiene la llave y en donde se dejará el registro leído.

Salidas:

longreg Longitud del registro leído. 0 si no se localizó ningún registro.

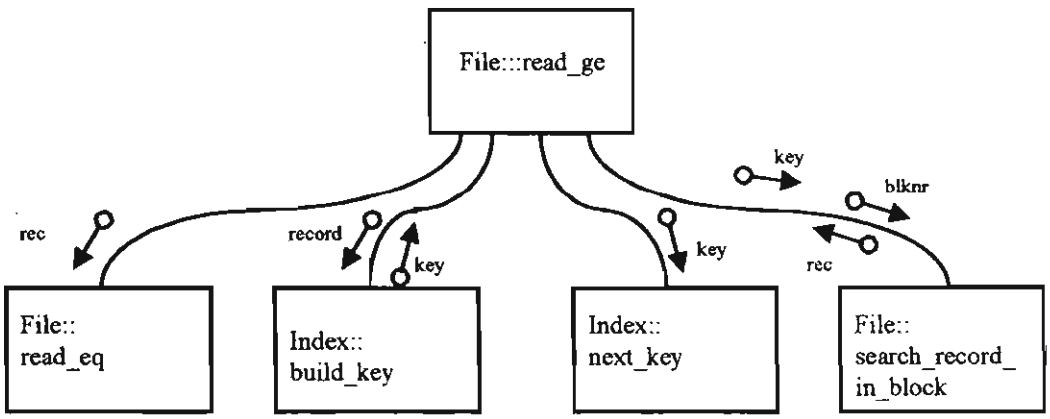
Proceso:

Extrae la llave del área del registro
Localiza la siguiente llave en el índice
Si no la localizó
    Regresa 0
Localiza el registro dentro del bloque de datos
Regresa la longitud del registro.

7.1.13 File::read_ge (Lee un registro con una llave mayor o igual)

Lee el registro con la menor llave que es mayor o igual a la llave contenida en el registro que se proporciona como argumento.

Diagrama de estructura:



Precondiciones:

La llave actual debe corresponder a la llave de referencia para la búsqueda.

Poscondiciones:

El registro leído se establece como registro actual.

Entradas:

rec	Apuntador al área que contiene la llave y en donde se dejará el registro leído.

Salidas:

Longreg	Longitud del registro leído. 0 si no se localizó ningún registro.

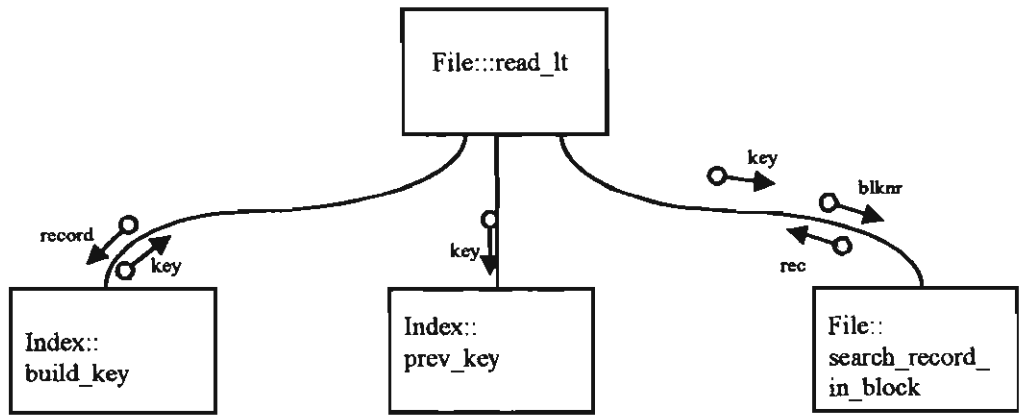
Proceso:

Lee un registro con llave igual
Si la lectura tuvo éxito
    Regresa el registro leído y su longitud
Extrae la llave del área del registro
Localiza la siguiente llave en el índice
Si no la localizó
    Regresa 0
Localiza el registro dentro del bloque de datos
Regresa la longitud del registro.

7.1.14 File::read_lt (Lee un registro con una llave menor)

Lee el registro con la mayor llave que es menor que la llave contenida en el registro que se proporciona como argumento.

Diagrama de estructura:



Precondiciones:

La llave actual debe corresponder a la llave de referencia para la búsqueda.

Poscondiciones:

El registro leído se establece como registro actual.

Entradas:

rec	Apuntador al área que contiene la llave y en donde se dejará el registro leído.

Salidas:

Longreg	Longitud del registro leído. 0 si no se localizó ningún registro.

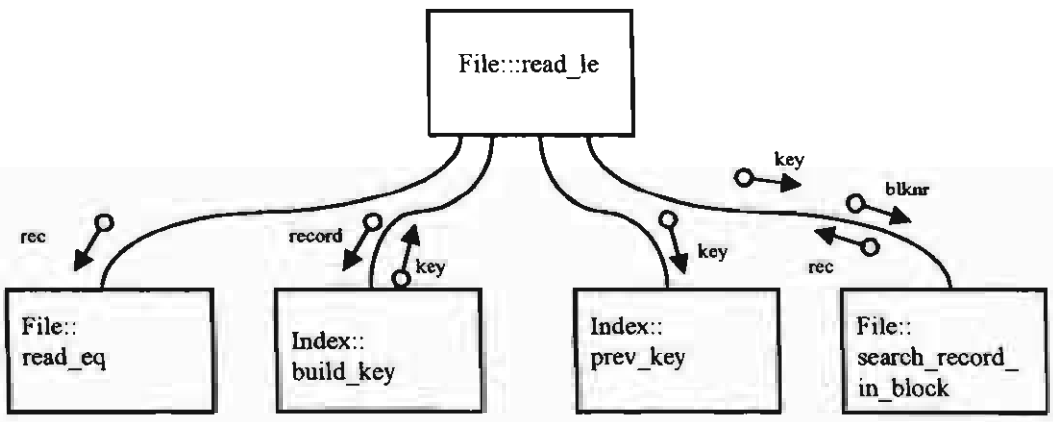
Proceso:

Extrae la llave del área del registro
Localiza la llave anterior en el índice
Si no la localizó
    Regresa 0
Localiza el registro dentro del bloque de datos
Regresa la longitud del registro.

7.1.15 File::read_le (Lee un registro con una llave menor o igual)

Lee el registro con la mayor llave que es menor o igual a la llave contenida en el registro que se proporciona como argumento.

Diagrama de estructura:



Precondiciones:

La llave actual debe corresponder a la llave de referencia para la búsqueda.

Poscondiciones:

El registro leído se establece como registro actual.

Entradas:

rec	Apuntador al área que contiene la llave y en donde se dejará el registro leído.

Salidas:

Longreg	Longitud del registro leído. 0 si no se localizó ningún registro.

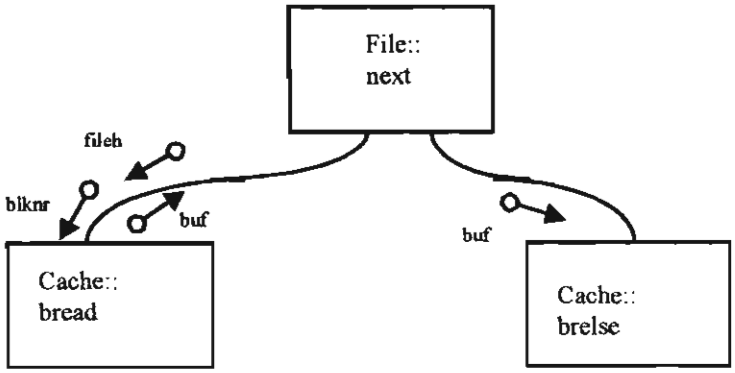
Proceso:

```
Lee un registro con llave igual
Si la lectura tuvo éxito
        Regresa el registro leído y su longitud
Extrae la llave del área del registro
Localiza la llave anterior en el índice
Si no la localizó
        Regresa 0
Localiza el registro dentro del bloque de datos
Regresa la longitud del registro.
```

7.1.16 File::next (Lectura secuencial física)

Esta función realiza una lectura secuencial física. Los bloques de datos se encuentran encadenados. En el bloque descriptor se encuentra el número de datos. Internamente se mantiene un apuntador al último registro leído. De esta manera es posible recorrer todos los bloques de datos del archivo sin utilizar las estructuras de índices. Normalmente el orden será el cronológico de inserción de los registros.

Diagrama de estructura:



Precondiciones:

La posición del registro actual debe haberse establecido.

Poscondiciones:

El registro actual será el siguiente.

Entradas:

No hay.

Salidas:

record El registro leído.

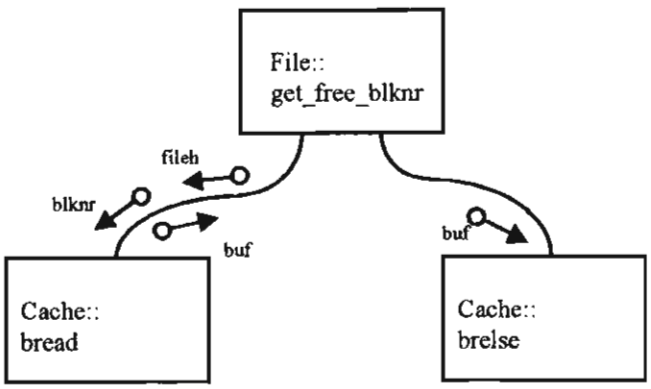
Proceso:

Si el bloque actual es cero
    Establece como bloque actual el primer bloque de datos del archivo
    Establece como registro actual el primer registro del bloque
sino
    Lee el bloque de datos actual
Si no hay más registros en el bloque de datos
    Libera el bloque de datos
    Lee el siguiente bloque de datos
    Establece como registro actual el primero del bloque
Mueve el registro del bloque de datos al registro (parámetro)
Establece como registro actual el siguiente registro del bloque
Libera el bloque de datos
Regresa la longitud del registro

7.1.17 File::get_free_blknr (Proporciona un número de bloque libre)

Esta función proporciona un número de bloque disponible dentro del archivo. El descriptor del archivo contiene el tamaño del archivo en bloques, entonces casi siempre este número será el siguiente bloque libre con lo cual se extenderá el archivo. Además, cuando se libera un bloque se añade a una lista de bloques libres para ser reutilizados posteriormente. Siempre que exista un bloque libre en la lista, este será el número asignado.

Diagrama de estructura:



Precondiciones:

No hay.

Poscondiciones:

El tamaño del archivo queda incrementado en un bloque

Entradas:

No hay.

Salidas:

blknr El número de bloque solicitado.

Proceso:

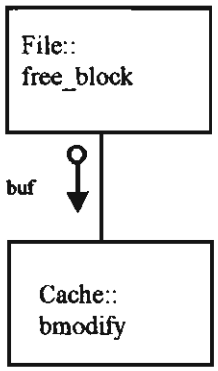
Si la lista de bloques liberados está vacía
    blknr = tamaño del archivo
    incrementa el tamaño del archivo
sino

blknr = el primer bloque de la lista de bloques liberados
Elimina el bloque de la lista
regresa blknr

7.1.18 File::free_block (Libera un bloque del archivo)

Cuando se elimina un registro de datos o se libera un bloque de índices el bloque vacío se incorpora a una lista de bloques libres para ser reasignado posteriormente.

Diagrama de estructura:



Precondiciones:

No hay.

Poscondiciones:

El bloque liberado queda al principio de la lista de bloques liberados.

Entradas:

buf Apuntador al buffer que contiene el bloque que se libera.

Salidas:

No hay.

Proceso:

Incorpora el bloque en la lista de bloques liberados
Marca el bloque liberado como modificado

7.2 Diseño funcional de la clase Index

7.2.1 Alineación de segmentos

Otro aspecto importante es el que se genera por el manejo de las llaves segmentadas. El usuario no tiene ninguna limitación respecto al tipo de datos que puede usar en cada segmento ni en el orden en que estos aparecen en la llave.

Para comparar correctamente los segmentos de dos llaves, éstos deben estar correctamente alineados de acuerdo al tipo de dato y la plataforma en que opera el sistema. En los bloques de índices es deseable maximizar su grado. Por esta razón, las llaves deben almacenarse empacadas sin los bytes de alineación.

En el sistema se creó un objeto Aligner que detecta en forma automática la alineación para cada tipo de dato dependiendo de la plataforma en que se encuentra operando y posteriormente presta sus servicios para agregar o eliminar los bytes de relleno de las llaves.

La clase Index es la responsable de mantener los árboles B que constituyen los índices. Contiene los siguientes elementos de información:

name. Nombre del índice. Es una cadena de hasta 9 caracteres que se utiliza como identificador del índice. Este nombre se utiliza en todas las futuras referencias al índice.

segnr. Número de segmentos que constituyen una llave para este índice.

Fstseg. Número del descriptor de segmentos en el bloque descriptor del archivo que contiene la descripción del primer segmento de la llave.

rootblk. Número de bloque dentro del archivo que corresponde al bloque raíz del árbol B.

keylen. Longitud de la llave en bytes. Es el espacio que ocupa la llave dentro de un bloque de índices. No contiene bytes de alineación.

keys_in_blk. El número máximo de llaves que puede contener un bloque de índices.

index_block_off. Desplazamiento del vector ib_index_blk en el bloque de índices. Es decir, a partir de esta posición se encuentra el vector de números de bloque de los bloques de índices del siguiente nivel.

data_blk_off. Desplazamiento del vector ib_data_blk en el bloque de índices. A partir de este byte se encuentra el vector de los números de bloques de datos que en

donde se encuentran los registros de datos que corresponden a las llaves que se encuentran en el bloque de índices.

key_off. Desplazamiento del vector de llaves en el bloque de índices.

Al descender en la estructura del árbol B es necesario dejar registrada la trayectoria del descenso, por ejemplo cuando se lee el archivo secuencialmente por índice se realiza un recorrido del árbol en orden interno. La trayectoria de la raíz a la hoja visitada queda registrada en los siguientes tres elementos de información:

curr_lev. Nivel actual. Es el nivel en el que se encuentra un algoritmo al descender en la estructura del árbol B. El nodo raíz tiene nivel 0.

curr_idx_blk. Es un vector que contiene los números de bloque de cada uno de los niveles que se han recorrido en el descenso.

curr_idx. Es un vector que contiene el índice del elemento del vector curr_idx_blk dentro de cada bloque de índices por donde se ha realizado el descenso.

En algunas operaciones es necesario salvar los tres valores antes descritos. Para esto se utilizan los siguientes datos.

save_lev. Salva el valor de curr_lev.

save_idx_blk. Salva el valor de curr_idx_blk.

save_idx. Salva el valor de curr:idx.

Finalmente se tiene los siguientes datos:

first_segment. Apuntador al primer elemento de la lista de objetos de la clase Segment que describen los segmentos de la llave.

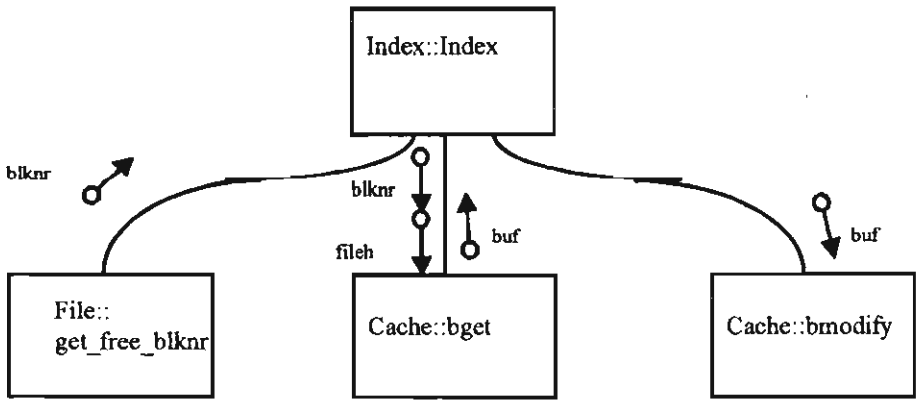
next. Apuntador al siguiente índice del archivo.
file. Apuntador al objeto file al que corresponde este índice.

7.3 Diseño funcional de la clase Index

7.3.1 Index::Index (Constructor)

Esta función tiene como objetivo la creación de las estructuras de datos que soportan la operación de un árbol B.

Diagrama de estructura:



Precondiciones:

No debe existir un índice con el mismo nombre.
Debe haber espacio en el bloque descriptor para almacenar el descriptor del índice y los descriptores de sus segmentos.

Poscondiciones:

No hay.

Entradas:

p_name	Nombre del índice. Este nombre será utilizado para las futuras referencias al índice. El nombre puede contener hasta nueve caracteres.
p_fstseg	Primer descriptor de segmentos que le corresponde en el bloque descriptor del archivo.
p_segnr	Número de segmentos que constituyen la llave.
seg	Vector de descriptores de los descriptores de segmentos que constituyen la llave.
p_file	Apuntador al objeto File al que pertenece el índice que se está creando.
p_rootblk	Número de bloque que constituye el bloque raíz del árbol B. Si este dato es cero, entonces se trata de un nuevo índice.

Salidas:

No hay

Proceso:

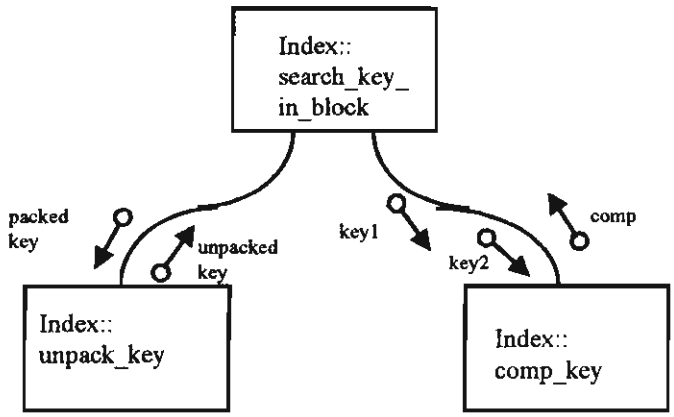
Si es la creación de un nuevo índice

Obtén un número de bloque del archivo
Obtén un buffer para el nuevo bloque
Inicializa el bloque con cero llaves
Marca el bloque como modificado
Inicializa los miembros del objeto Index
Por cada segmento de la llave
Crea un objeto de la clase Segment
Copia los datos del vector seg
Incrementa la longitud de la llave con la longitud del segmento
Calcula los parámetros de manipulación de bloques de índices
keys_in_block Número máximo de llaves en un bloque de índices
index_blk_off Desplazamiento del vector de números de bloques de índices
Data_blk_off Desplazamiento del vector de números de bloques de datos
key_off Desplazamiento del vector de llaves

7.3.2 Index::search_key_in_block (Localiza una llave en un bloque de índices)

Realiza la búsqueda de una llave en un bloque de índices. La función recibe como argumentos un bloque de índices y una llave desempacada. La función busca la llave en el bloque en forma secuencial. Para poder comparar las llaves del bloque de índices es necesario desempacarlas. La función regresa la posición en donde se encuentra la llave o donde le correspondería estar si no se encuentra. Además la función establece el valor de la variable global exist_in_block.

Diagrama de estructura:



Precondiciones:

No hay.

Poscondiciones:

No hay.

Entradas:

index_blk	Bloque de índices.
unp_key	Llave desempacada para búsqueda del registro.

Salidas:

posición	Posición de la llave dentro del bloque (donde esta o debiera estar)
exist_in_block	1. Si se encontró la llave; 0. Si no se encontró la llave

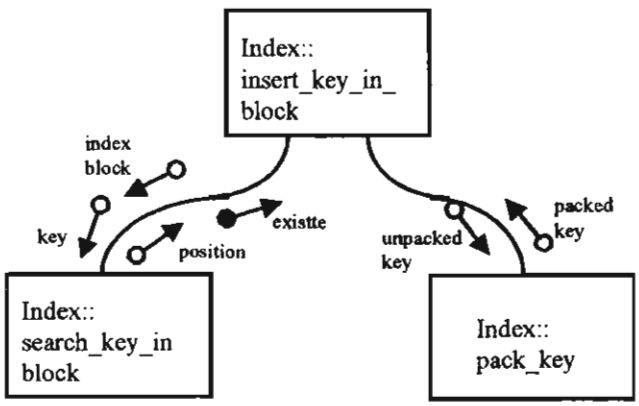
Proceso:

```
Por cada una de las llaves en el bloque de índices
        Desempaca la llave en blk_key
        Si unp_key = blk_key
                exist_in_block = true
                Regresa la posición de la llave
        Si unp_key < blk_key
                salir
exist_in_block = false
regresa la posición
```

7.3.3 Index::insert_key_in_block (Inserta una llave en un bloque de índices)

Una llave con su número de bloque de datos asociado se inserta en un bloque de índices. Debe existir espacio en el bloque para la nueva llave y ésta no debe aparecer en el bloque. Se determina la posición de la llave. Las llaves se recorren a la derecha para abrir espacio para la nueva llave.

Diagrama de estructura:



Precondiciones:

Debe haber espacio en el bloque para agregar la llave.

Poscondiciones:

La nueva llave queda en la posición que le corresponde.

Entradas:

ib	Bloque de índices en donde se va a insertar la llave.
key	Llave que se va a insertar.
data_blknr	Número del bloque de datos en donde está el registro al que corresponde la llave.
prom_blknr	Número de bloque de índices que se inserta asociado con la llave.

Salidas:

No hay.

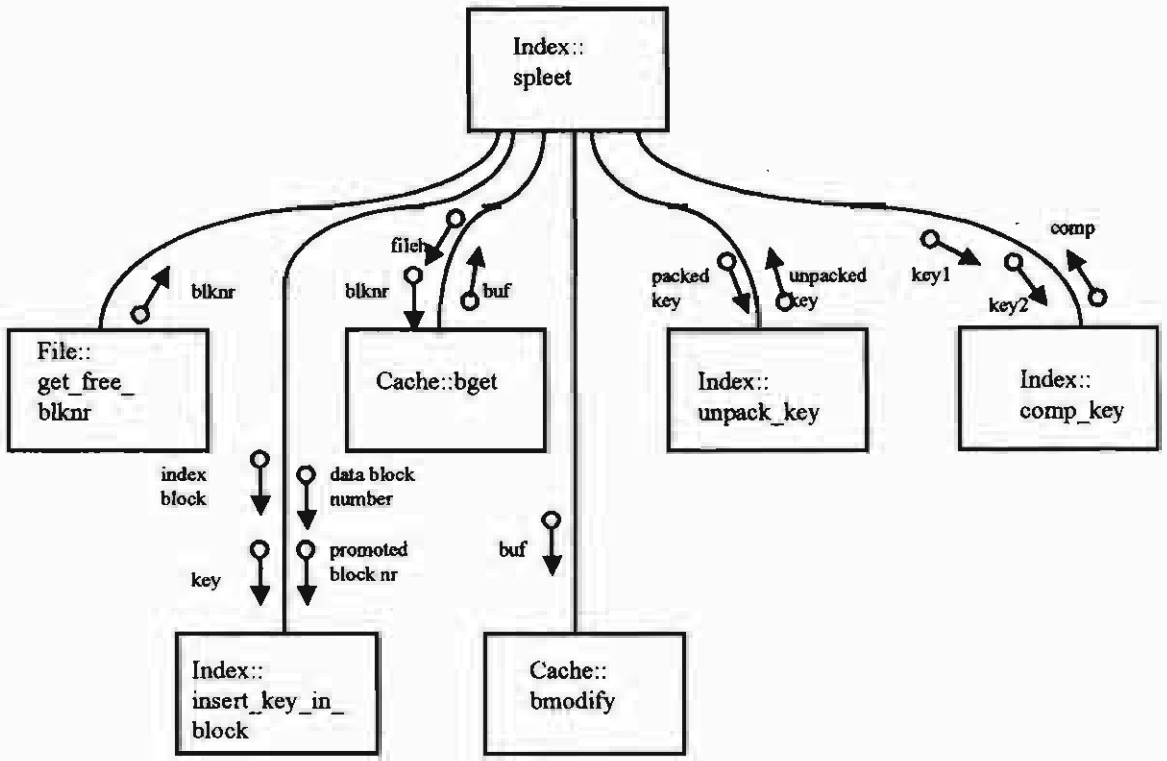
Proceso:

Verifica la precondición
Busca la llave en el bloque
Si existe
    Error
Recorre a la derecha las llaves que son mayores
Mueve la llave a su posición dentro del bloque de índices
Incrementa el número de índices dentro de bloque de índices

7.3.4 Index::spleet (Divide bloque de índices)

Cuando se requiere insertar una llave en un bloque que se encuentra lleno se produce una condición de desborde (overflow). Esa función crea un nuevo bloque y distribuye equitativamente las llaves en ambos bloques. La nueva llave se agrega en el bloque con menos llaves. La función regresa la llave que debe ser promovida al siguiente nivel en la estructura de índices.

Diagrama de estructura:



Precondiciones:

El bloque de índices se encuentra totalmente lleno.

Poscondiciones:

Las llaves quedan distribuidas en dos bloques de índices.
Se promueve la llave media para que se inserte en el nivel inmediato superior.

Entradas:

ib	Bloque de índices lleno
prom_key	Llave que se va a inserta
data_blknr	Número de bloque de datos asociado a la llave

Salidas:

prom_key	Llave promovida
data_blknr	Número de bloque de datos asociado a la llave
prom_blknr	Número de bloque de índices promovido

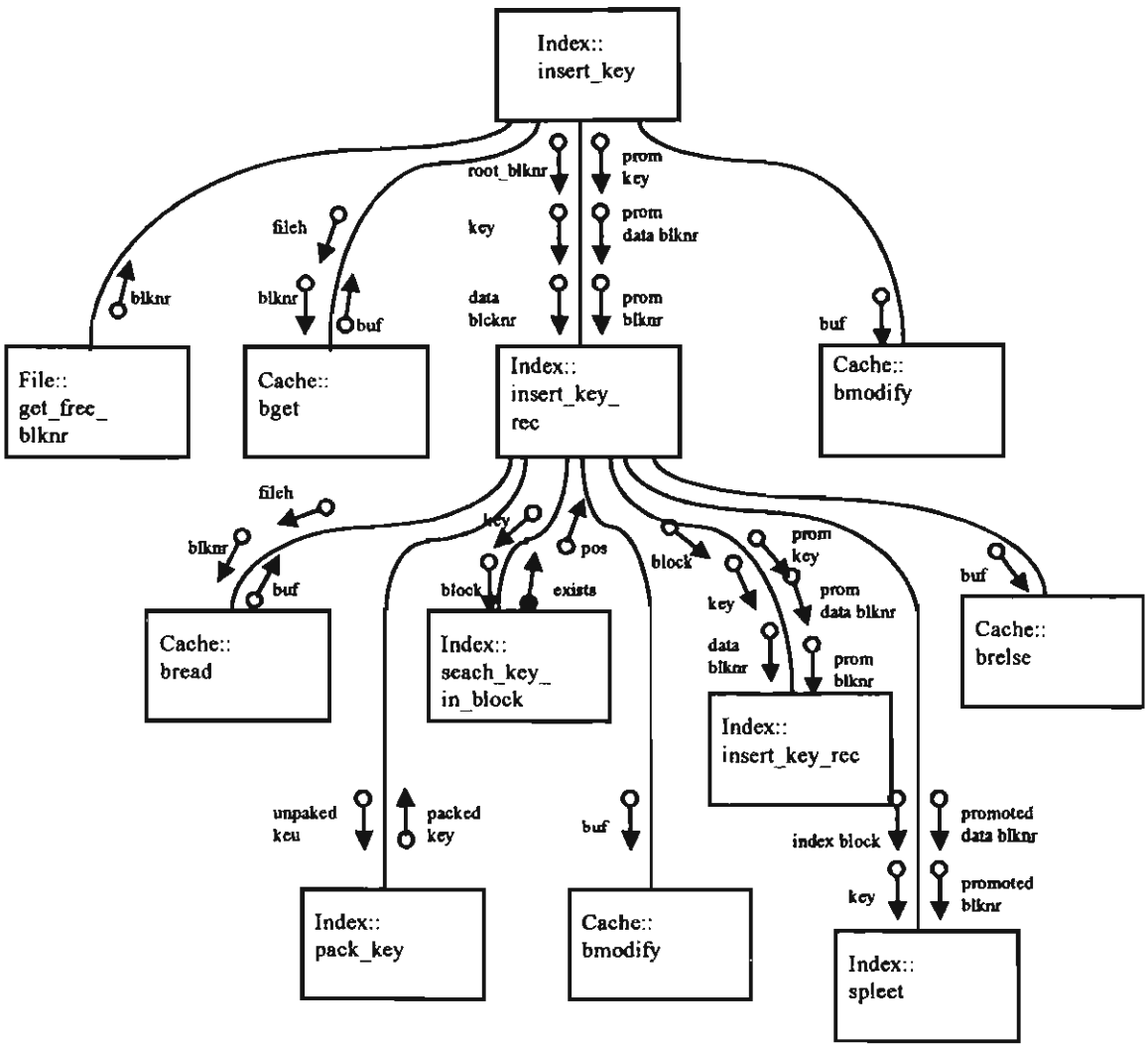
Proceso:

Calcula mk = número de llaves que quedarán en el viejo bloque
Calcula nmk = número de llaves que se mandarán al nuevo bloque
Obtén un número de bloque para el nuevo bloque
Obtén el buffer para el nuevo bloque
Mueve nmk bloques del viejo bloque al nuevo bloque
Si prom_key > última llave del viejo bloque
    Inserta la llave en el nuevo bloque
sino
    Inserta la llave en el viejo bloque
Si el viejo bloque tiene más llaves que el nuevo bloque
    Promueve la última llave del viejo bloque
sino
    Promueve la primera llave del nuevo bloque
Marca el nuevo bloque como modificado

7.3.5 Index::insert_key (Inserta una llave en la estructura de índices)

La función recibe como argumentos la llave y su número de bloque asociado. La función invoca a File::insert_key_rec, la cual realiza la tarea de inserción en forma recursiva, es decir, en su descenso se llega hasta el nivel de las hojas del árbol B y en su ascenso se procesan las llaves promovidas (en caso de que se produzcan). Finalmente, si en el bloque raíz se produce una promoción, entonces se crea un nuevo nivel de índices.

Diagrama de estructura:



Precondiciones:

No hay.

Poscondiciones:

La llave queda insertada en el árbol.

Entradas:

key	Llave que va a ser insertada.
dbnr	Número del bloque de datos asociado a la llave.

Proceso:

insert_key_rec(blknr_root, key, dbnr, prom_key, prom_dbnr, prom_blknr)
Si se genera una promoción (no hay espacio ni en el bloque raíz)
Crea un nuevo bloque raíz en la estructura de índices
Coloca en el nuevo bloque raíz la llave promovida
El número de bloque de la raíz anterior
La llave promovida y número de bloque de datos asociado
El número de bloque de índice promovido
Marca el nuevo bloque de índices como modificado
Actualiza el número del bloque raíz con el nuevo bloque

7.3.6 Index::insert_key_rec (Inserta llave recursivamente)

Al insertar una llave es necesario descender primeramente hasta el nivel de las hojas y tratar de insertar en una hoja la nueva llave. Si la hoja en donde debiera encontrarse la llave se encuentra llena, se debe producir un **spleet,** es decir, partir el bloque de índices (esto lo hace la función spleet) y promover una llave al nivel anterior de la estructura de índices. Esto se hace al regresar de las recursiones en forma ascendente hasta llegar al bloque raíz.

Diagrama de estructura:

Se incluyó con la función anterior a fin de dar claridad al diagrama.

Precondiciones

No hay.

Poscondiciones:

No hay.

Entradas:

curr_blknr	Número de bloque de índices en donde se va a insertar la llave.
key	Llave que se va a insertar.
dbnr	Número de bloque de datos asociado a la llave.

Salidas:

prom_key	Llave promovida (si se produce promoción).
prom_dbnr	Número de bloque de datos asociado a la llave promovida.
prom_blknr	Número de bloque de índices promovido

valor 0. No hubo promoción; 1. Si hubo promoción

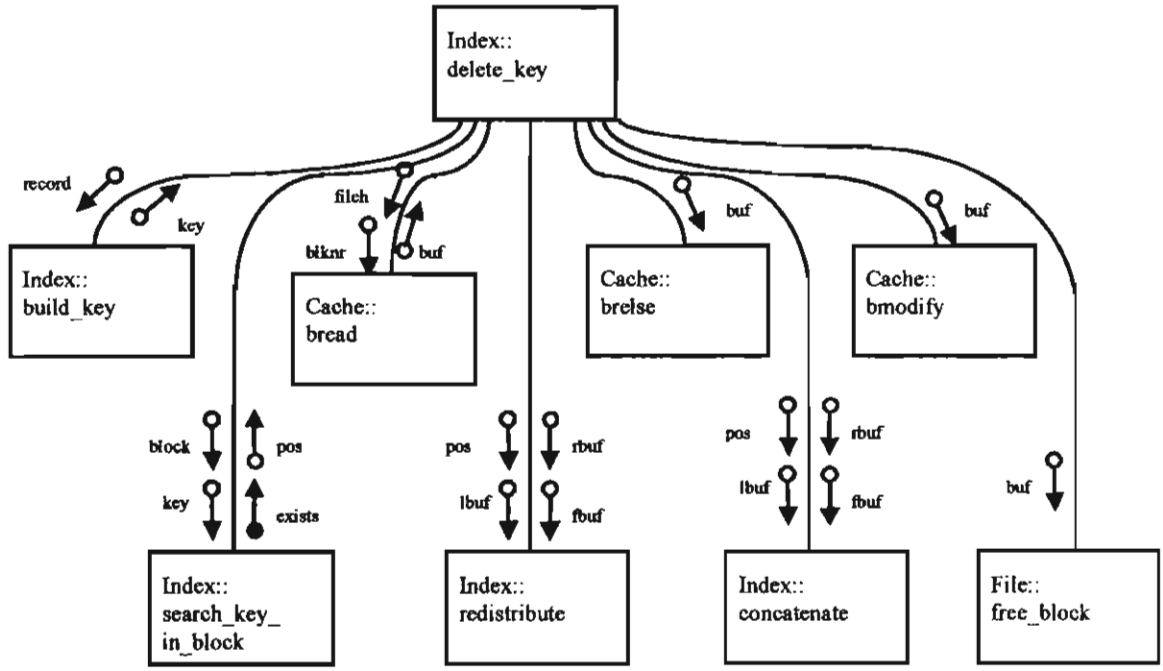
Proceso:

```
Si nos pasamos de nivel de las hojas (curr_blknr = 0)
        Promueve la llave
                prom_key = key
                prom_dbnr = dbnr
                prom_blknr = 0
        Regresa 1 (hubo promoción)
Lee el bloque de datos curr_blknr
Determina la posición de la llave en el bloque (donde está o podría estar)
Si existe la llave
        Error
ret = insert_key_rec(número de bloque[pos], key, dbnr,
                prom_key, prom_dbnr, prom_blknr)
Si no hubo promoción
        Libera el bloque leído
        Regresa 0 (no hubo promoción)
// Hubo promoción del siguiente nivel
Si hay espacio en el bloque de índices
        Agrega la llave promovida al bloque
                prom_key
                prom_dbnr
                prom_blknr
        Marca el bloque como modificado
        Regresa 0
spleet(bloque actual, prom_key, prom_dbnr, prom_blknr)
Marca el bloque como promovido
Return 1 (hubo promoción)
```

7.3.7 Index::delete_key (Elimina una llave de un índice)

Al eliminar una llave de la estructura de índices es posible provocar una condición de deficiencia (underflow) en alguno de los bloques. La función verifica si esta condición se ha presentado y se auxilia de las funciones File::concatenate y File::redistribute para mantener las restricciones del árbol B y mantenerlo balanceado.

Diagrama de estructura:



Precondiciones:

No hay.

Poscondiciones:

La llave desaparece de la estructura del índice.

Entradas:

record Registro de datos conteniendo los segmentos de la llave a eliminar.

Salidas:

valor 1. La llave se eliminó; 0. La llave no se eliminó.

Proceso:

El vector blknr[ ] almacena la ruta de descenso desde el bloque de índices raíz
El vector pos[ ] almacena las posiciones por donde se realizó el descenso

nivel = 0
blknr [nivel]= Número del bloque de índices raíz

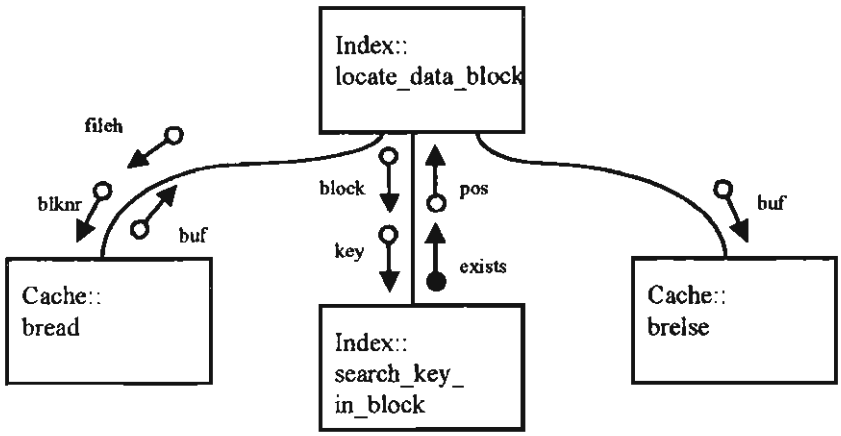
```
Construye la llave tkey con la información del registro
Hacer
        Lee bloque de índices blknr[nivel]
        Localiza posición i de la llave tkey
        pos[nivel] = i
        Si se encontró la llave
                Salir
        nivel = nivel + 1
        blknr[nivel] = apuntador al siguiente nivel[i]
        Libera el bloque de índices
Mientras blknr[nivel] no sea cero
Si no se localizó la llave
        Error
// Si la llave se encuentra en un nodo interior se sustituye por la llave sucesora
//      que se encuentra descendiendo primeramente por el hijo derecho y después
//      por el primer hijo de cada nodo

hijo pos[nivel] + 1 (el hijo derecho)
Mientras el bloque de índices sea interior
        nivel = nivel +1
        pos[nivel] = 0
        blknr[nivel] = número de bloque de índices[hijo]
        hijo = 0 (primer hijo)
        Si el bloque actual no es el que contiene la llave
                Libera el bloque de índices
        Lee bloque de índices blknr[nivel]
Si el bloque (ib) donde se encuentra la llave es interior
        Mueve la llave inmediata (primera en bloque hib) a la posición de la llave a
        eliminar
        Bloque ib = bloque hib (pasamos a la hoja)
Elimina la llave del bloque ib (una hoja)
Reduce el número de llaves del bloque
Mientras el nivel no sea cero y tengamos un bloque con deficiencia de llaves
        Lee el bloque del nivel anterior fbuf (blknr[nivel-1])
        Determina los bloques hermanos vecinos izquierda (lblknr) y derecha (rblknr)
        Si hay bloque hermano izquierdo
                Lee el bloque hermano izquierdo lbuf
                Redistribuye con el hermano izquierdo
                Si la redistribución tuvo éxito
                        Marca modificados bloque padre y hermano izquierdo
                        Salir
                sino
                        Libera bloque hermano izquierdo
```

```
Si no se ha realizado la redistribución
        Lee el bloque hermano derecho
        Redistribuye con el hermano derecho
        Si la redistribución tuvo éxito
                Marca como modificados los bloques padre y hermano
derecho
                Salir
        sino
                Libera el bloque hermano derecho
    // No fue posible hacer la redistribución. Procede concatenación
    Si hay bloque hermano derecho
        Lee el bloque hermano izquierdo
        Concatena con el bloque izquierdo
        Libera el bloque original
        Marca como modificado el bloque izquierdo
        Desciende un nivel en el árbol
    sino
        Lee el bloque hermano derecho
        Concatena con el bloque derecho
        Libera el bloque original
        Libera el bloque derecho
        Marca modificado el bloque original
        Desciende un nivel en el árbol
Marca como modificado el bloque original
Regresa el número de bloque actual
```

7.3.8 Index::locate_data_bock (Localiza bloque de datos)

Esta función tiene como propósito localizar el bloque de datos correspondiente a una llave. La función utiliza el índice actual para localizar el bloque de datos.

Diagrama de estructura:



Precondiciones:

No hay.

Poscondiciones:

La posición del índice queda registrada en el estado del archivo.

Entradas:

key	Llave cuyo bloque de datos asociado se intenta localizar.

Salidas:

dbnr	Número de bloque de datos asociado a la llave.

Proceso:

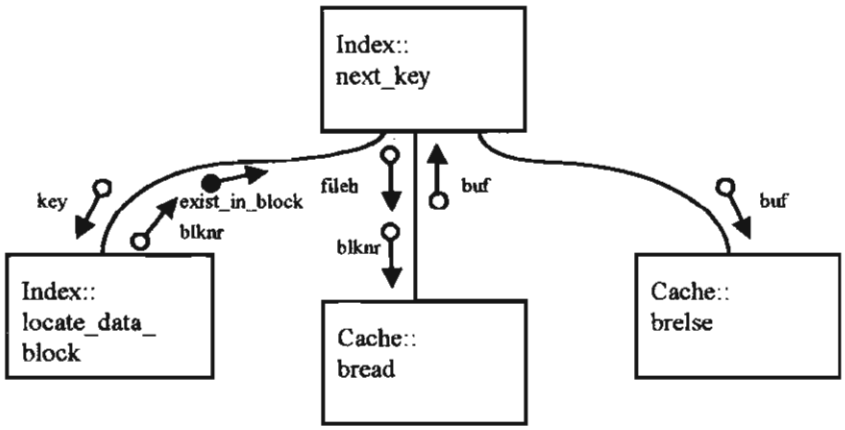
```
curr_lev = 0
blknr = número del bloque raíz del índice actual
curr_idx_blknr[curr_lev] = blknr
curr_idx[curr_lev] = 0
Mientras blknr no sea 0
        Lee el bloque de índices blknr
        Determina la posición de la llave en el bloque
        curr_idx[curr_lev] = posición
        Si la llave está en el bloque
                dbnr = data_blknr[posición]
```

```
                Libera el bloque de índices
                Regresa dbnr
        Si hay un siguiente nivel de índices
                curr_lev = curr_lev + 1
                blknr = index_blknr[posición]
                curr_idx_blk[curr_lev] = blknr
                Libera el bloque de índices
        sino
                Libera el bloque de índices
                Regresa o
Regresa blknr
```

7.3.9 Index::next_key (Localiza la siguiente llave)

A partir de una llave argumento, localiza la siguiente llave en la estructura del índice actual. La función invoca a la función Index::locate_data_block la cual desciende en la estructura de índices hasta llegar al bloque que contiene la llave (o donde la llave podría estar) dejando la ruta de descenso en los vectores curr_idx_blk y curr_idx que son los números de bloque de cada nivel de índices y la posición dentro del bloque por la cual se realizó el descenso respectivamente.

Diagrama de estructura:



Precondiciones:

El índice actual debe corresponder a la llave que se proporciona como argumento.

Poscondiciones:

No hay.

Entradas:

key	Llave a partir de la cual se va a realizar la búsqueda de la siguiente llave.

Salidas:

key	Valor de la llave localizada.
valor	Número del bloque de datos en donde se encuentra el registro que corresponde a la llave localizada. Si no se localizó una llave, entonces se regresa un cero.

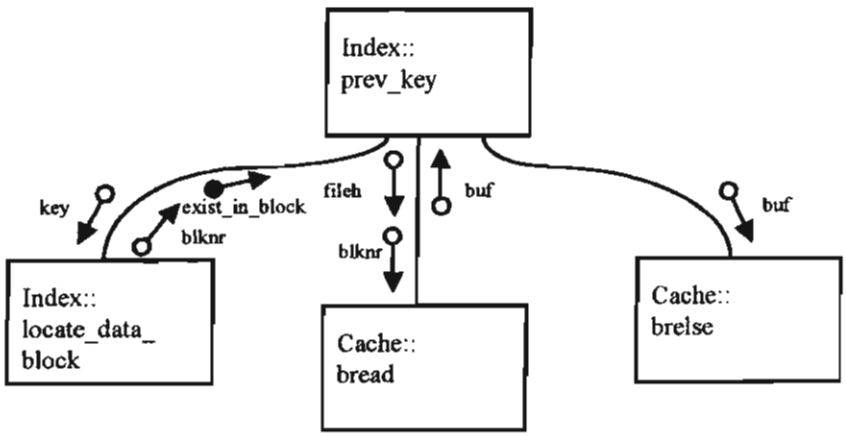
Proceso:

```
Localiza bloque de datos
Lee el bloque de índices de mayor nivel
Si la llave se encuentra o debiera encontrarse en una hoja
        Si existe la llave
                Posicionate en la siguiente posición del bloque
        Si existe una llave en la posición
                Mueve la llave de esa posición al argumento
                blknr = número de bloque de datos en esa posición
                Libera el bloque
                Regresa blknr
        Mientras no sea el bloque raíz y no haya más llaves en el bloque
                Libera el bloque
                Reduce el nivel del árbol
                Lee el bloque por donde se realizó el descenso
        Si no hay más llaves después de la posición de descenso
                Regresa 0 (No existe una llave mayor)
        Mueve la llave de la posición de descenso al argumento
        blknr = núnero de bloque de datos en la posición de descenso
        Libera el bloque de índices
        Regresa blknr
Sino    // La llave se encontró en un bloque intermedio
        blknr = número de bloque de índices siguiente en el bloque
        Mientras no se tenga una hoja
                Libera el buffer
                Lee el bloque blknr
                blknr = primer número de bloque de índices
        Mueve la primera llave al argumento
        blknr = primer número de bloque de datos
        Libera el bloque
        Regresa blknr
```

7.3.10 Index::prev_key (Localiza la llave anterior)

A partir de una llave de referencia se localiza la llave anterior en la estructura del índice. En esta función, como en la anterior, primeramente se invoca a la función Index::locate_data_block.

Diagrama de estructura:



Precondiciones:

El índice actual debe corresponder a la llave que se proporciona como argumento.

Poscondiciones:

No hay.

Entradas:

key	Apuntador a la llave que se utiliza como referencia.

Salidas:

key	El valor de la nueva llave sustituye al argumento.
valor	El número del bloque de datos en donde se encuentra el registro asociado a la llave. Cero si no existe una llave menor que el argumento.

Proceso:

Localiza el registro que corresponde a la llave

```
Lee el último bloque de índices utilizado para localizar el registro
Si el bloque es una hoja del árbol B
        Mientras no sea el bloque raíz y la posición de descenso no sea cero
                Libera el bloque
                Desciende de nivel
                Lee el bloque que corresponde a la ruta de descenso
        Si la posición de descenso es cero
                Libera el buffer
                Regresa 0 (no existe una llave menor)
        Mueve la llave de la posición anterior a la posición de descenso
        blknr = número de bloque de datos asociado a la llave localizada
        Libera el bloque de índices
        Regresa blknr
Sino
        blknr = número de bloque de índices en la posición de descenso
        Mientras no sea una hoja del árbol B
                Libera el bloque
                Lee el bloque número blknr
                blknr = último número de bloque de índices en el bloque
        blknr = último número de bloque de datos en el bloque de índices
        Mueve la última llave del bloque al argumento
        Libera el bloque
        Regresa blknr
```

7.3.11 Index::redistribute (Redistribuye las llaves en dos bloques)

Redistribuye equitativamente las llaves contenidas en dos bloques del mismo nivel para que tengan más o menos el mismo espacio libre. En el bloque padre de ambos se debe cambiar la llave media para conservar la estructura del árbol B.

Precondiciones:

No hay.

Poscondiciones:

Los dos bloques quedan con un número similar de llaves.

Entradas:

pos	Posición en el bloque padre en donde deberá quedar la llave media.
lbuf	Buffer que contiene el bloque hijo de la izquierda.

fbuf	Buffer que contiene el bloque padre.
rbuf	Buffer que contiene el bloque hermano derecho.

Proceso:

Si entre los dos bloques no tienen al menos tantas llaves como para llenar un bloque
    Regresa FALSO (no procede la redistribución)
Calcula pro = promedio de bloques entre lbuf y rbuf
Si en el bloque izquierdo hay menos llaves que su grado d
    Calcula mk = número de llaves que hay que mover del bloque derecho al izquierdo
    Mueve la llave[pos] del bloque padre al final del bloque izquierdo
    Mueve llave[0] a llave[mk-2] del principio del bloque derecho al final del izquierdo
    Llave[pos] del bloque padre = llave[mk-1] del bloque derecho
    Ajusta a la izquierda las llaves que quedaron en bloque derecho
    Ajusta el número de llaves en los dos bloques
    Regresa 1 (OK)
// El bloque derecho tiene déficit de llaves
Calcula mk = número de llaves a mover del bloque izquierdo al derecho
Recorre las llaves del bloque derecho mk posiciones a la derecha
Mueve la llave[pos] del bloque padre al bloque derecho
Mueve las últimas mk -1 llaves del bloque izquierdo al bloque derecho
Mueve la última llave que quedó en el bloque izquierdo a llave[pos] del bloque padre
Ajusta el número de llaves de los bloques izquierdo y derecho
Regresa 1 (OK)

7.3.12 Index::Concatenate (Concatena dos bloques de índices)

Cuando dos bloques adyacentes del mismo nivel no contienen el grado mínimo de llaves, el contenido de los dos bloques se concentra en uno sólo y el otro se libera.

Precondiciones:

La suma de los números de llaves en ambos bloques es menor o igual a keys_in_blk

Poscondiciones:

Todas las llaves se concentran en el bloque hijo izquierdo.
El bloque padre queda con una llave menos.
El bloque hijo derecho queda sin llaves.

Entradas:

pos	Posición en el bloque padre de la llave que se tiene que ajustar.
lbuf	Buffer que contiene el bloque hijo izquierdo.
fbuf	Buffer que contiene el bloque padre
rbuf	Buffer que contiene el bloque hijo derecho

Salidas:

No hay.

Proceso:

Mueve llave[pos] del bloque padre al final del bloque izquierdo
Mueve todas las llaves del bloque hijo derecho al final del bloque hijo izquierdo
Elimina la llave[pos] del bloque padre
Ajusta el número de llaves en el bloque izquierdo
Reduce el número de llaves del bloque padre

7.3.13 Index::build_key (Construye una llave a partir de un registro)

Cuando el usuario necesita enviar una llave al manejador para que este realice alguna operación como lectura aleatoria o posicionamiento con respecto a un índice, lo hace colocando los segmentos de la llave dentro de un registro de datos en las posiciones que les corresponde. El sistema utiliza la función File:build_idx para armar la llave con la información contenida en el registro de datos.

Precondiciones:

El índice actual debe haberse establecido con la función File::index.
El registro de datos debe contener los segmentos de la llave.

Poscondiciones:

Se obtiene una llave concatenada.

Entradas:

record	Registro de datos con los segmentos de la llave.

Salidas:

key Llave expandida (con bytes de relleno para alinear los segmentos)

Proceso:

Por cada uno de los segmentos de la llave
Alinea el destino con respecto al tipo de dato
Mueve el segmento del área del registro de datos a la llave
Incrementa el apuntador a la llave en la longitud del segmento

7.3.14 Index::unpack_key (Desempaca una llave)

Las llaves se encuentran empacadas dentro de los bloques de índices (sin bytes de relleno) y los segmentos de las llaves no se encuentran alineados respecto al tipo de dato. Esta función toma una llave empacada y la expande a fin de poder compararla con otra. Al compararla se debe comparar uno a uno los segmentos de datos y estos deben estar alineados de acuerdo al tipo de dato.

Precondiciones:

El índice actual debe de estar definido.

Poscondiciones:

No hay.

Entradas:

packed Llave empacada.

Salidas:

key Llave desempacada.

Proceso:

Por cada uno de los segmentos de la llave
Alinea key de acuerdo al tipo de dato.
Mueve el segmento de la llave de paked a key
Incrementa packed con la longitud del segmento
Incrementa key con la longitud del segmento

7.3.15 Index::pack_key (Empaca una llave)

Esta función realiza la operación contraria de la función anterior. Se utiliza para almacenar las llaves en los bloques de índices.

Precondiciones:

Debe establecerse el índice actual.

Poscondiciones:

No hay.

Entradas:

key Llave desempacada.

Salidas:

packet Llave empacada.

Proceso:

Por cada uno de los segmentos de la llave
Alinea la llave desempacada key
Mueve el segmento de la llave desempacada a la llave empacada
Incrementa los apuntadores a las dos llaves con la longitud del segmento

7.3.16 Index::comp_key (Compara dos llaves desempacadas)

Con esta función se realizan todas las comparaciones de llaves. Ya que una llave puede estar compuesta por varios segmentos de diferentes tipos de datos y el orden del segmento puede ser ascendente o descendente.

Precondiciones:

Debe establecerse el índice actual.

Poscondiciones:

No hay.

Entradas:

key1, key2 Apuntadores a las dos llaves que se van a comparar.

Salidas:

valor 0. Llaves iguales; >0. key1 > key2; <0. key1 < key2

Proceso:

Por cada uno de los segmentos de la llave
    Alinea los apuntadores de las dos llaves de acuerdo al tipo de segmento
    Compara los dos segmentos de acuerdo al tipo de datos
    Si son iguales
        Continua
    sino
        Regresa el resultado de la comparación de los segmentos
Regresa 0

7.3.17 Index::del (Elimina la estructura de un índice)

Libera recursivamente todos los bloques que constituyen los nodos del árbol B de un índice. Los bloque liberados se incorporan a la lista de bloques libres del archivo para posterior reutilización.

Diagrama de contexto:

n/a (no aplicable)

Precondiciones:

No hay.

Poscondiciones:

No hay.

Entradas:

blknr El número de bloque a partir del cual se va a hacer la eliminación.

Salidas:

No hay.

Proceso:

```
Lee el bloque blknr
Por cada uno de los números de bloques del siguiente nivel
        Si el número de bloque es cero
                salir de la iteración (estamos en una hoja)
        Elimina recursivamente el bloque
Libera el bloque de disco
```

7.4 El diseño funcional de la clase Database

La clase Database es una clase con persistencia lo que permite conservar el estado de la base de datos entre una sesión y otra. La información contenida en el objeto Database es la siguiente:

num_file. Número de archivos File_db que conforman la base de datos.

num_sets. Número de sets que conforman la base de datos.

path. Trayectoria del archivo que contiene la definición de la base de datos. En este archivo se almacena la información del objeto de la clase Database.

first_file. Apuntador al primer elemento de una lista de estructuras file_node en la que se registran todos los archivos que constituyen la base de datos. La estructura file_node contiene los siguiente elementos:

file_db. Apuntador a un objeto de la clase File_db.

path. Trayectoria de acceso al archivo File_db.

next. Apuntador al siguiente nodo de la lista.

first_set. Apuntador al primer elemento de una lista de estructuras set_node en la que se registran todos los sets que constituyen la base de datos. La estructura set_node contiene los siguientes elementos:

set. Apuntador a un objeto de la clase Set.

path. Trayectoria de acceso al set.

next. Apuntador al siguiente nodo de la lista.

fileh. Asa del archivo que contiene la definición de la base de datos.

A continuación se describen las funciones miembro de la clase Database.

7.4.1 Database::Database (Constructora)

Al crear un objeto del tipo Database el sistema identifica si ya existe la base de datos en el disco. Si ya existe la información simplemente la carga, y si no existe, la crea.

Precondiciones:

No las hay

Poscondiciones:

Todos los archivos y conjuntos que constituyen la base de datos quedan abiertos

Entradas:

path La trayectoria en donde reside la base de datos

Salidas:

No hay.

Proceso:

Copia el path de la base de datos
Si no existe la base de datos
    Inicializa la base de datos como vacía
    Regresa
Abre el archivo que contiene la base de datos en disco
Lee el número de archivos
Lee el número de sets
Por cada archivo en la base de datos
    Crea un nuevo nodo en la lista de archivos
    Lee la longitud del path del archivo
    Adquiere memoria para el path

Lee el path del disco
Crea el archivo y coloca su apuntador en el nodo de la lista de archivos
Marca el archivo como abierto
Por cada set en la base de datos
Crea un nodo para la lista de sets
Lee el número de archivo que le corresponde al archivo propietario
Localiza el archivo propietario en la lista de archivos
owner = apuntador al archivo propietario
Lee el número de archivo que le corresponde al archivo miembro
Localiza el archivo miembro en la lista de archivos
memb = apuntador al archivo miembro
Lee la longitud del path del set
Adquiere memoria para el path
Lee el path del set
Crea el set y coloca su apuntador en el nodo de la lista de sets
Abre el set
Conecta el set con la base de datos
Marca el set como abierto

7.4.2 Database::~Database (Destructora)

La función destructora serializa la base de datos. Almacena en el disco una estructura equivalente para su posterior recuperación.

Precondiciones:

No hay.

Poscondiciones:

No hay

Entradas:

No hay.

Salidas:

No hay.

Proceso:

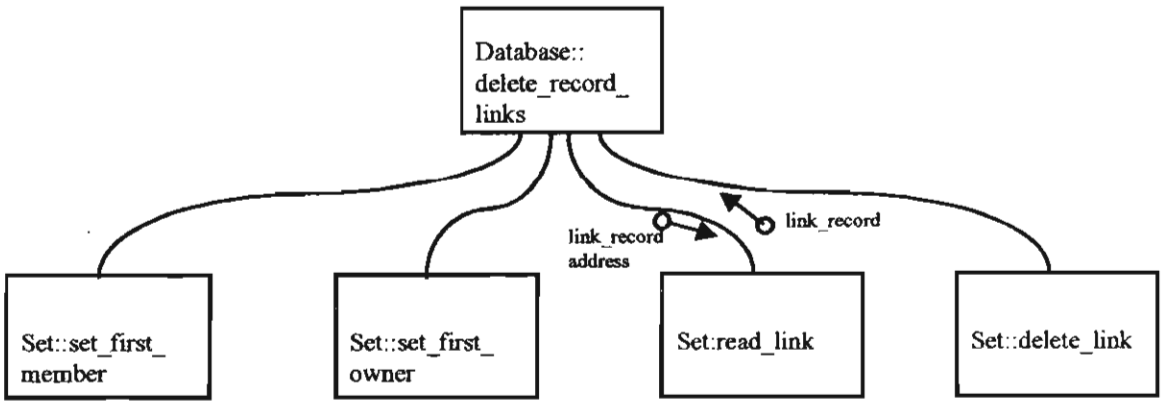
Abre el archivo en donde se almacenará la base de datos
Graba el número de archivos

Graba el número de sets
Por cada uno de los archivos
    Graba la longitud del path del archivo
    Graba el path del archivo
Por cada uno de los sets
    Determina el número secuencial que le corresponde al archivo propietario
    Graba el número secuencial que le corresponde al archivo propietario
    Determina el número secuencial que le corresponde al archivo miembro
    Graba el número secuencial que le corresponde al archivo miembro
    Graba la longitud del path del set
    Graba el path del set
Por cada archivo
    Elimina el archivo
    Elimina el path del archivo
Por cada set
    Elimina el set
    Elimina el path del set

7.4.3 Database::delete_record_links (Elimina los eslabones que apuntan a un registro)

La función delete_record_links tienen como propósito la eliminación de todos los eslabones de todas las cadenas en donde se hace referencia a un registro de un archivo que se ha dado de baja.

Diagrama de estructura :



Como puede apreciarse la función Set::link_start no requiere parámetros ya que el objeto Set mantiene información de cuál es el último registro procesado. Tampoco regresa información, ya que solamente altera su estado interno.

Precondiciones:

No hay.

Poscondiciones:

Todos los eslabones de todos sets en donde se hace referencia a un registro son eliminados.

Entradas:

file_db Apuntador al archivo del que se está eliminando un registro

Salidas:

No hay.

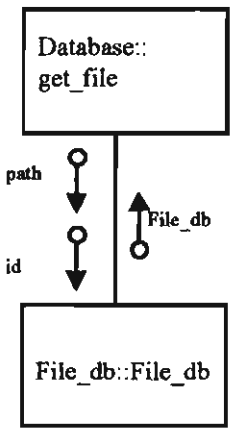
Proceso:

Por cada uno de los conjuntos de la base de datos
Si file_db figura como propietario del set
Inicializa la cadena del propietario
Elimina todos los eslabones
Por cada uno de los sets de la base de datos
Si file_db figura como miembro del set
Inicializa la cadena del miembro
Elimina todos los eslabones

7.4.4 Database::get_file (Crea u obtiene un archivo)

Database es la clase responsable de administrar los archivos que constituyen una base de datos. Por esta razón la función Database::get_file los proporciona.

Diagrama de estructura:



Precondiciones:

No hay.

Poscondiciones:

El archivo queda registrado en la base de datos.

Entradas:

path Trayectoria del archivo que se desea crear o simplemente obtener (si ya existe).

Salidas:

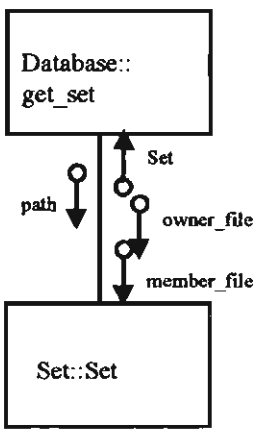
file_db Apuntador al archivo solicitado.

Proceso:

Si el archivo se encuentra en la lista de archivos
    Regresa un apuntador al archivo
sino
    Crea un nuevo nodo al final de la lista de archivos
    Crea el archivo
    Coloca el apuntador al archivo en el nuevo nodo
    Almacena el path del archivo en el nuevo nodo
    Incrementa el número de archivos
    Conecta el archivo con la base de datos

7.4.5 Database::get_set (Crea u obtiene un set)

Diagrama de estructura:



Precondiciones:

No hay.

Poscondiciones:

El set solicitado queda registrado en la base de datos.

Entradas:

path	Trayectoria del set que se solicita.
owner_file	Apuntador al archivo propietario del set.
member_file	Apuntador al archivo miembro del set.

Salidas:

set	Apuntador al set solicitado.

Proceso:

```
Si el set se encuentra en la lista de sets
        Regresa un apuntador al set
sino
        Crea un nuevo nodo al final de la lista de sets
        Crea el set
        Coloca el apuntador al set en el nuevo nodo
        Almacena el path del set en el nuevo nodo
        Conecta el set con la base de datos
```

7.5 Diseño funcional de la clase File_db

La clase File_db es heredera de la clase File por lo que contiene la misma información pero además cuando se crea un objeto de la clase File_db se genera una llave primaria que consiste de un campo de tipo Serial definido en las primeras posiciones del registro. La otra diferencia en el sistema es que los objetos de la clase File_db no pueden declararse de manera independiente, sino deben ser solicitados a un objeto de la clase Database.

7.5.1 File_db::File_db (Constructora)

La función de creación de un archivo de la base de datos invoca al constructor de la clase File, de la que se deriva. Adicionalmente crea una llave serial para el archivo. Esto es un

objeto File_db es lo mismo que un objeto File al que se ha adicionado una llave Serial en los primeros cuatro bytes del archivo.

Precondiciones:

Debe existir un objeto Database.

Poscondiciones:

No hay.

Entradas:

file_path Ruta del archivo.

Salidas:

No hay.

Proceso:

Invoca al constructor de la clase File.
Crea un índice serial al principio del registro.

7.5.2 File_db::del (Elimina un registro)

Esta función es similar al la función File::del, pero antes de eliminar el registro, se invoca a la función Database::delete_record_links, la cual se encarga de eliminar todas las referencias al registro en todos los sets que constituyen la base de datos.

7.6 Diseño funcional de la clase Set

La clase Set es la responsable del manejo de los conjuntos que constituyen una base de datos. La clase Set se deriva de la clase File, sin embargo, la herencia es de carácter privado ya que la funcionalidad heredada de File no se pone disponible al usuario, es decir, un usuario no puede leer ni grabar registros en la clase Set. En vez de esto, en la clase Set se definen una serie de funciones para que el usuario pueda manipular los conjuntos de una manera consistente y controlada a fin de mantener su integridad.

7.6.1 Set::Set (Constructora)

Esta función solamente puede ser invocada por la clase Database, ya que es la clase que controla la base de datos.

Precondiciones:

No hay.

Poscondiciones:

No hay.

Entradas:

path	Ruta y nombre del archivo que contiene el set.
owner_file	Apuntador al archivo File_db propietario del set.
member_file	Apuntador al archivo File_db miembro del set.

Salidas:

No hay.

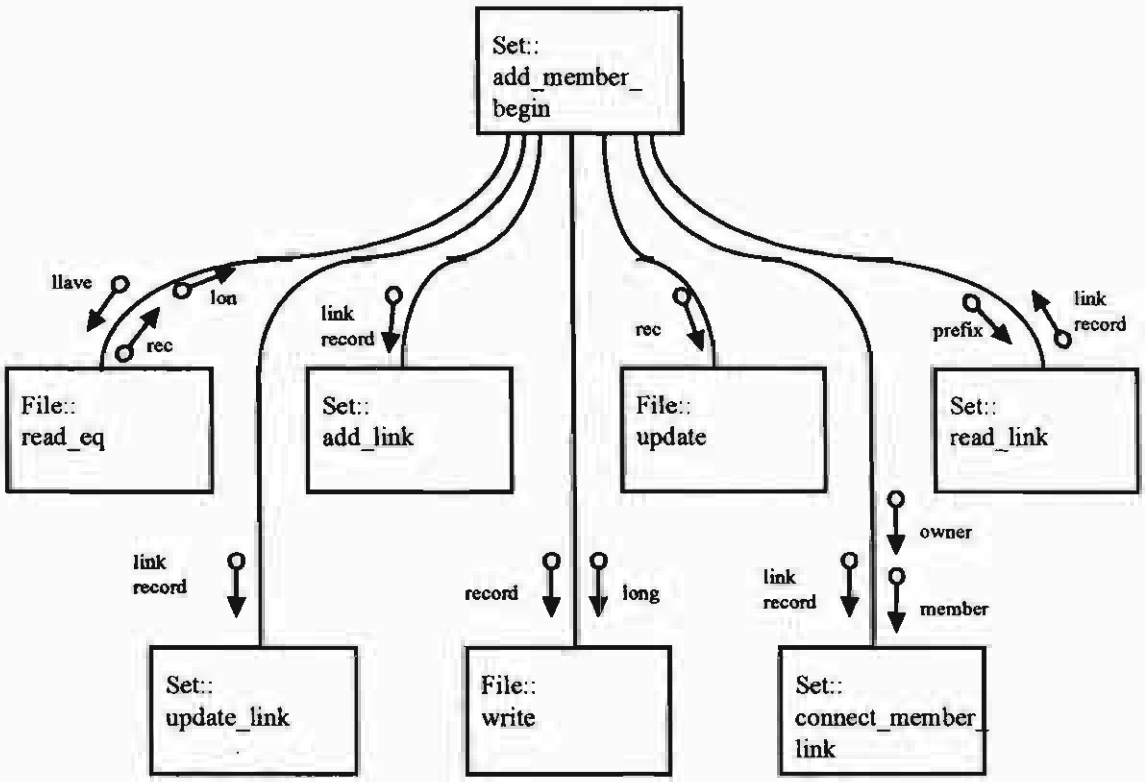
Proceso:

Invoca al constructor de la clase File
Registra en el objeto los archivos propietario y miembro del set
Inicializa el eslabón actual a nulo
Si el set no existe
    Crea un índice para los registros anclas
Establece el índice creado como índice actual

7.6.2 Set::add_member_begin (Agrega un registro miembro al principio del set)

Se incorpora un eslabón en la cadena que parte del propietario con una instancia del registro miembro y una instancia del registro propietario. También se incorpora un eslabón en la cadena de la instancia del registro miembro.

Diagrama de estructura:



Precondiciones:

No hay.

Poscondiciones:

El eslabón insertado queda como eslabón actual en la cadena.

Entradas:

owner	Identificador serial de la ocurrencia del registro propietario.
member	Identificador serial de la ocurrencia del registro miembro.

Salidas:

No hay.

Proceso:

Lee el ancla que le corresponde al registro propietario
Crea un nuevo eslabón con los datos del registro propietario y del registro miembro

Si existe el ancla

Inserta el eslabón al principio de la cadena existente
Actualiza el ancla la cual apuntará al nuevo eslabón insertado
Si ya existía un eslabón previamente

Lee el eslabón que ya existía
Actualiza su referencia al eslabón anterior

sino

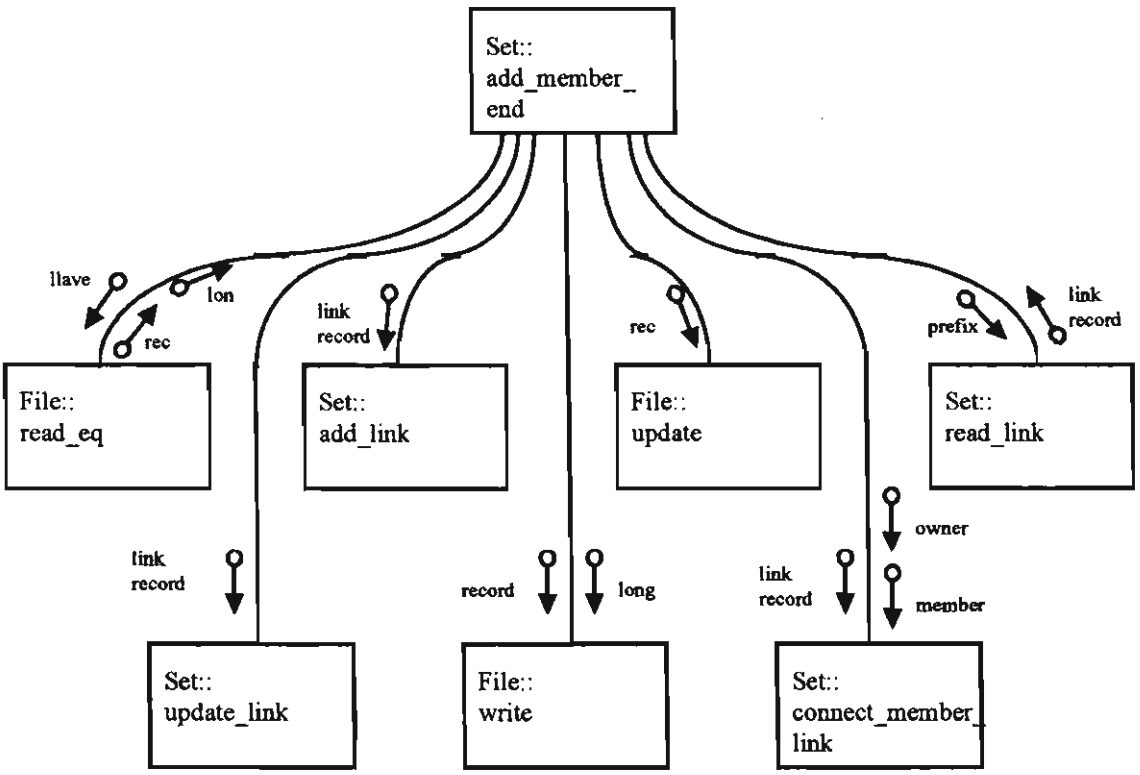
Agrega el eslabón
Crea un registro ancla para el propietario

Inserta un eslabón en la cadena del registro miembro

7.6.3 Set::add_member_end (Agrega un registro miembro al final del set)

Realiza la inserción de un nuevo eslabón al final de la cadena del registro propietario.

Diagrama de estructura:



Poscondiciones:

El eslabón insertado queda como eslabón actual en la cadena.

Entradas:

owner	Identificador serial de la ocurrencia del registro propietario.

member Identificador serial de la ocurrencia del registro miembro.

Salidas:

No hay.

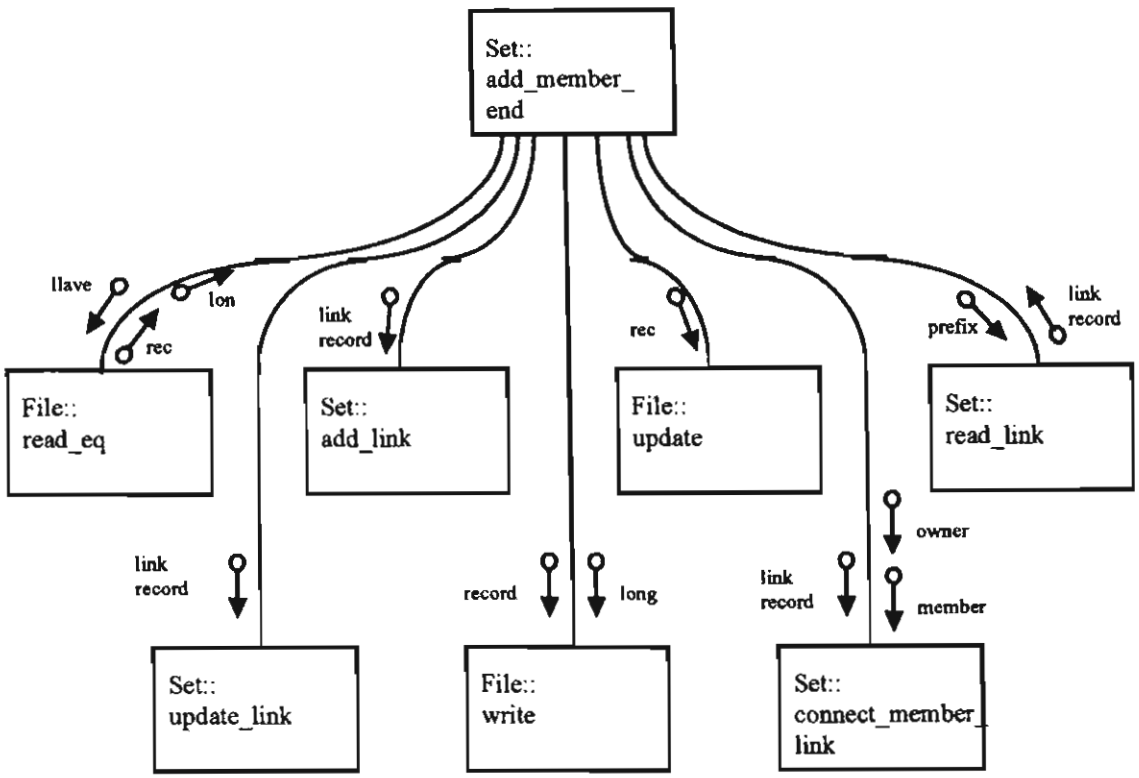
Proceso:

Lee el ancla que le corresponde al registro propietario
Crea un nuevo eslabón con los datos del registro propietario y del registro miembro
Si existe el ancla
    Inserta el eslabón al final de la cadena existente
    Actualiza el ancla la cual apuntará al nuevo eslabón insertado
    Si ya existía un eslabón previamente
        Lee el eslabón que ya existía
        Actualiza su referencia al eslabón anterior
sino
    Agrega el eslabón
    Crea un registro ancla para el propietario
Inserta el eslabón en la cadena del registro miembro

7.6.4 Set::connect_member_link (Conecta un eslabón en la cadena miembro)

Conecta el eslabón que se acaba de agregar, a la cadena del registro miembro. Con esto, el eslabón queda conectado con todos los registros propietarios del mismo registro miembro.

Diagrama de estructura:



Precondiciones:

No hay.

Poscondiciones:

El eslabón forma parte de dos cadenas. La del propietario con todos sus miembros y la del miembro con todos sus propietarios.

Entradas:

owner	Identificador del registro propietario.
member	Identificador del registro miembro.
lr	Apuntador a un registro eslabón que ya se encuentra en la cadena propietario.

Salidas:

No hay.

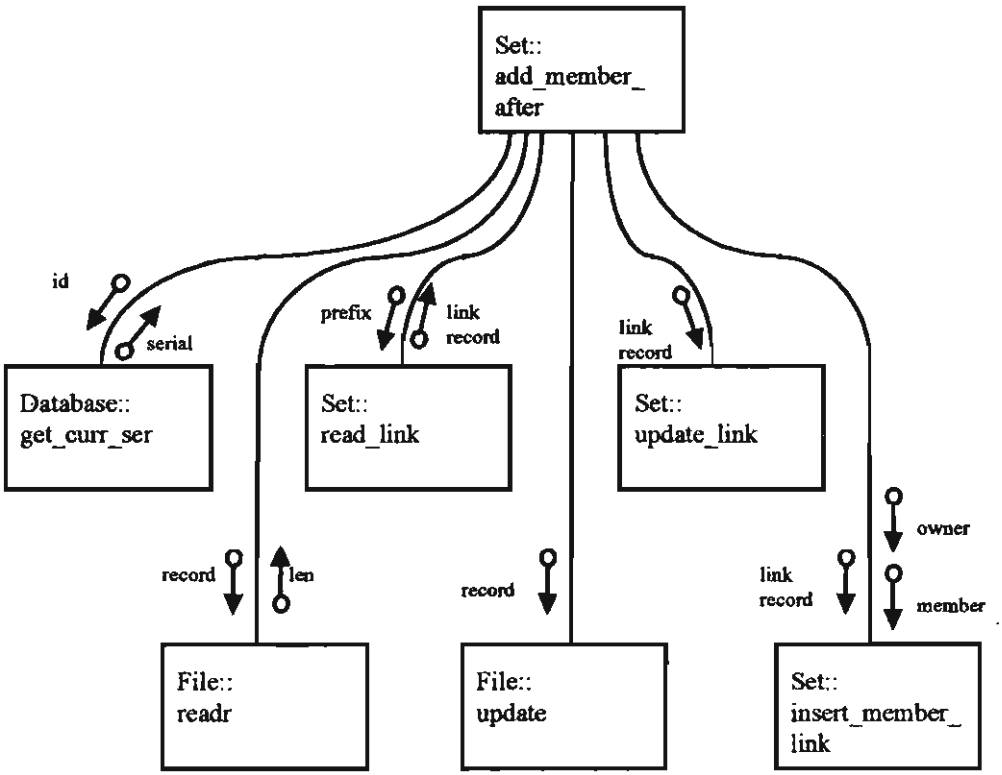
Proceso:

Lee el ancla con el identificador del registro miembro
Si existe el ancla
    Lee el primer eslabón de la cadena miembro (olr)
    Actualiza su apuntador previo. Apunta al eslabón que se incorpora (lr)
    Actualiza el apuntador siguiente de lr para que apunte a olr
    Actualiza el ancla para que apunte a lr
sino
    Crea un registro ancla
    El ancla apunta a lr como primer eslabón

7.6.5 Set::add_member_after (Agrega un registro miembro después del actual)

Todo set contiene un elemento llamado actual_link que apunta al eslabón que hace referencia al registro miembro más recientemente procesado por el sistema. El nuevo miembro se inserta inmediatamente después del registro miembro actual.

Diagrama de Estructura:



Precondiciones:

No hay.

Poscondiciones:

No hay.

Entradas:

No hay.

Salidas:

No hay.

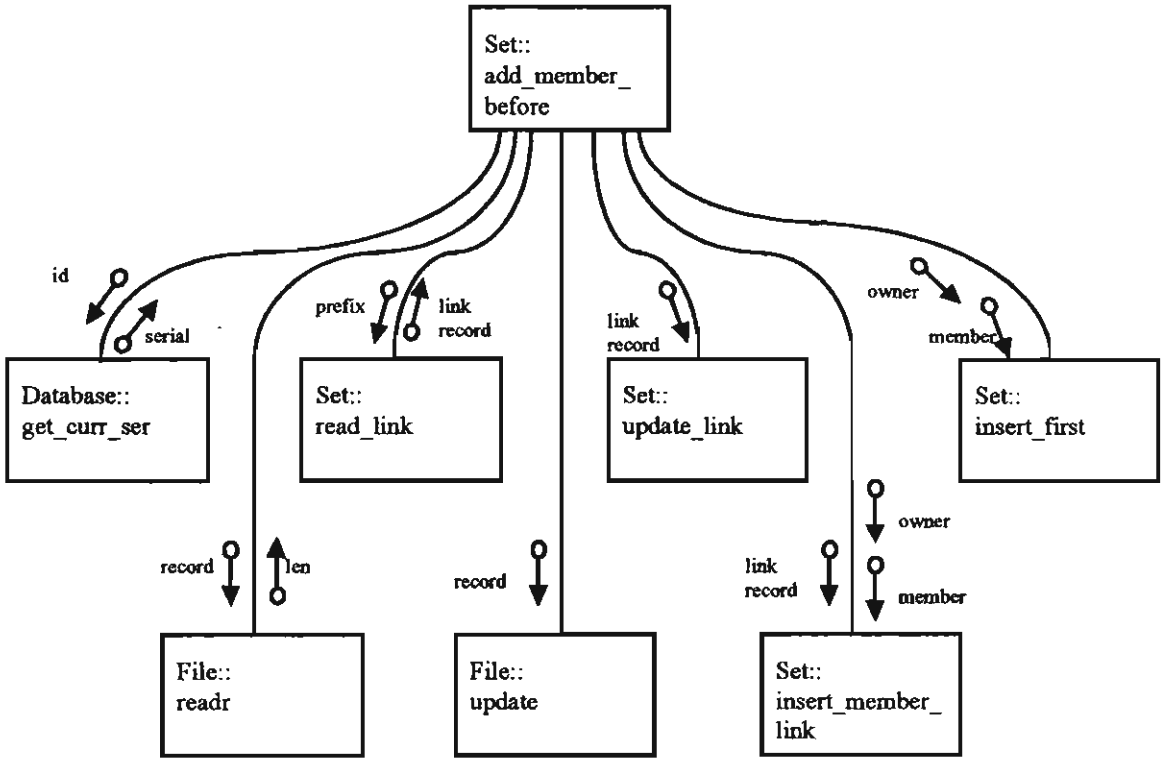
Proceso:

Obtiene los identificadores seriales del registro actual propietario y del registro actual miembro
Si el eslabón actual está definido y pertenece al mismo registro propietario
Lee el eslabón actual
Crea un nuevo eslabón con los identificadores de los registros propietario y miembro
Encadena el nuevo registro después del eslabón actual.
Si existe un eslabón posterior al actual
Actualiza su apuntador al anterior para apuntar al nuevo eslabón
sino
Actualiza el apuntador al **último** del ancla para que apunte al nuevo eslabón
Inserta el eslabón en la cadena del registro miembro
sino
Inserta el eslabón como primer eslabón de la cadena propietario

7.6.6 Set::add_member_before (Agrega un registro miembro antes del actual)

Todo set contiene un elemento llamado actual_link que apunta al eslabón que hace referencia al registro miembro más recientemente procesado por el sistema. El nuevo miembro se inserta inmediatamente antes del registro miembro actual.

Diagrama de estructura:



Precondiciones:

No hay.

Poscondiciones:

No hay.

Entradas:

No hay.

Salidas:

No hay.

Proceso:

Obtiene el identificador serial del registro actual en el archivo propietario
Obtiene el identificador serial del registro actual en el archivo miembro
Construye un nuevo eslabón **new_lk** con los registros propietario y miembro

Si el eslabón actual pertenece al mismo registro propietario
    Inserta el nuevo eslabón en la cadena antes del eslabón actual
    Si el eslabón actual era el primero de la cadena
        Actualiza el apuntador **primero** del ancla
    Inserta el eslabón en la cadena del registro miembro
sino
    Inserta el eslabón como primer eslabón de una nueva cadena del registro propietario

7.6.7 Set::add_link (Graba un eslabón)

Esta función graba un eslabón de una cadena en un bloque de datos del set. Para los eslabones no se generan índices en el archivo.

Precondiciones:

No hay.

Poscondiciones:

No hay:

Entradas:

Ir Registro eslabón que se va a grabar en el archivo.

Salidas:

Prefix Dirección en el disco de la ubicación en donde quedó grabado el eslabón.

Proceso:

Si es el primer registro de datos que se graba en el archivo
    Asigna un nuevo bloque de datos
    Inicializa el nuevo bloque de datos
sino
    Lee el último bloque de datos del archivo
Si no hay lugar en el bloque de datos para el eslabón
    Obtén un nuevo bloque de datos
    Encadena el nuevo bloque con el último
Mueve el eslabón al bloque de datos
Actualiza el espacio utilizado

Marca el bloque como modificado
Regresa la dirección en disco en donde se registró el nuevo eslabón

7.6.8 Set::delete_link (Elimina un eslabón)

Precondiciones:

El eslabón a eliminar es el eslabón actual.

Poscondiciones:

No hay.

Entradas:

No hay.

Salidas:

No hay.

Proceso:

```
Si no existe un eslabón actual
        Regresa.

// Ajustes en la cadena del propietario
Si hay un eslabón siguiente en la cadena del propietario
        Actualiza el apuntador anterior en el eslabón siguiente
Si hay un eslabón anterior en la cadena del propietario
        Actualiza el apuntador siguiente en el eslabón anterior
Si no hay eslabón anterior en la cadena del propietario
        Lee el ancla
        Si no hay eslabón siguiente en la cadena del propietario
                Elimina el ancla
        sino
                Actualiza el apuntador al primer eslabón en el ancla
sino
        Si no hay eslabón siguiente
                Lee el ancla
                Actualiza el apuntador al último eslabón del ancla
```

// Ajustes en la cadena del registro miembro
Si hay un eslabón siguiente en la cadena del miembro
    Actualiza el apuntador anterior en el eslabón siguiente
Si hay un eslabón anterior en la cadena del registro miembro
    Actualiza el apuntador siguiente en el eslabón anterior
Si no hay eslabón anterior en la cadena del registro miembro
    Lee el ancla
    Si no hay eslabón siguiente en la cadena del registro miembro
        Elimina el ancla
    sino
        Actualiza el apuntador al primer eslabón en el ancla
sino
    Si no hay eslabón siguiente
        Lee el ancla
        Actualiza el apuntador al último eslabón del ancla

7.6.9 Set::set_first_member (Establece el primer registro miembro como actual)

Se establece para el registro actual del archivo propietario del set como eslabón actual el primer eslabón de la cadena.

Precondiciones:

No hay.

Poscondiciones:

El eslabón actual queda establecido.

Entradas:

No hay.

Proceso:

Obtén el identificador serial del registro actual del archivo propietario
Lee el ancla del registro propietario
Si existe el ancla
    Establece como eslabón el primero apuntado por el ancla
sino
    Establece como nulo el eslabón actual

7.6.10 Set::set_last_member (Establece el último registro miembro como actual)

Se localiza el último registro miembro del registro propietario acutal y se establece como registro miembro actual.

Precondiciones:

No hay.

Poscondiciones:

El eslabón actual queda establecido.

Entradas:

No hay.

Proceso:

Obtén el identificador serial del registro actual del archivo propietario
Lee el ancla del registro propietario
Si existe el ancla
    Establece como eslabón el último apuntado por el ancla
sino
    Establece como nulo el eslabón actual

7.6.11 Set::set_first_owner(Establece el primer registro propietario como actual)

Se localiza el primer registro propietario del registro miembro actual del set y se establece como registro propietario actual.

Precondiciones:

No hay.

Poscondiciones:

El primer eslabón de la cadena de registros miembros queda establecido como actual.

Entradas:

No hay.

Proceso:

Obtén el identificador serial del registro actual del archivo miembro
Lee el ancla del registro miembro
Si existe el ancla
    Establece como eslabón el primero apuntado por el ancla
sino
    Establece como nulo el eslabón actual

7.6.12 Set::set_last_owner(Establece el primer registro propietario como actual)

Se establece para el registro actual del archivo miembro del set como eslabón actual el primer eslabón de la cadena del registro miembro.

Precondiciones:

No hay.

Poscondiciones:

El último eslabón de la cadena de propietarios queda establecido como actual.

Entradas:

No hay.

Proceso:

Obtén el identificador serial del registro actual del archivo miembro
Lee el ancla del registro miembro
Si existe el ancla
    Establece como eslabón actual el último apuntado por el ancla
sino
    Establece como nulo el eslabón actual

7.6.13 Set::get_next_member (Obtén el siguiente registro miembro del set)

Con esta función se navega entre los diferentes registros miembros pertenecientes en el set al mismo registro propietario.

Precondiciones:

El eslabón actual debe estar definido.

Poscondiciones:

El eslabón actual será el siguiente de la cadena.

Entradas:

No hay.

Salidas:

rec	Registro miembro del set leido
valor	Longitud del registro leido

Proceso:

Obtén el identificador serial del registro propietario actual
Si el registro actual es diferente del último propietario usado en el set
    Inicializa el eslabón actual para el nuevo registro propietario
Si el eslabón actual es nulo
    regresa 0
Lee el eslabón actual **lr**
Establece como eslabón actual el siguiente de la cadena
Mueve el serial del registro miembro del eslabón **lr** al área del registro
Lee el registro miembro
Regresa la longitud del registro

7.6.14 Set::prev_member (Lee el miembro anterior en la cadena propietario)

Con esta función se navega hacia atrás en los diferentes registros miembros pertenecientes en el set al mismo registro propietario.

Precondiciones:

El eslabón actual debe estar definido.

Poscondiciones:

El eslabón actual será el siguiente de la cadena.

Entradas:

No hay.

Salidas:

rec	Registro miembro del set leído
valor	Longitud del registro leído. Cero si no hay más registros.

Proceso:

Obtén el identificador serial del registro propietario actual
Si el registro actual es diferente del último propietario usado en el set
    Inicializa el eslabón actual con el último para el nuevo registro propietario
Si el eslabón actual es nulo
    regresa 0
Lee el eslabón actual **lr**
Establece como eslabón actual el anterior de la cadena
Mueve el serial del registro miembro del eslabón **lr** al área del registro
Lee el registro miembro
Regresa la longitud del registro

7.6.15 Set::get_next_owner (Obtén el siguiente registro propietario)

Con esta función se navega entre los diferentes registros propietarios pertenecientes en el set al mismo registro miembro.

Precondiciones:

El eslabón actual debe estar definido.

Poscondiciones:

El eslabón actual será el siguiente de la cadena.

Entradas:

No hay.

Salidas:

rec	Registro miembro del set leído
valor	Longitud del registro leído. Cero si no hay más registros.

Proceso:

Obtén el identificador serial del registro miembro actual

Si el registro actual es diferente del último miembro usado en el set
    Inicializa el eslabón actual para el nuevo registro miembro
Si el eslabón actual es nulo
    regresa 0
Lee el eslabón actual **lr**
Establece como eslabón actual el siguiente de la cadena
Mueve el serial del registro propietario del eslabón **lr** al área del registro
Lee el registro propietario
Regresa la longitud del registro

7.6.16 Set::get_prev_owner (Obtén el anterior registro propietario)

Con esta función se navega entre los diferentes registros propietarios pertenecientes en el set al mismo registro miembro.

Precondiciones:

El eslabón actual debe estar definido.

Poscondiciones:

El eslabón actual será el siguiente de la cadena.

Entradas:

No hay.

Salidas:

rec	Registro miembro del set leído
valor	Longitud del registro leído. Cero si no hay más registros.

Proceso:

Obtén el identificador serial del registro miembro actual
Si el registro actual es diferente del primer miembro usado en el set
    Inicializa el eslabón actual para el nuevo registro miembro
Si el eslabón actual es nulo
    regresa 0
Lee el eslabón actual **lr**
Establece como eslabón actual el anterior de la cadena
Mueve el serial del registro propietario del eslabón **lr** al área del registro
Lee el registro propietario
Regresa la longitud del registro

7.6.17 Set::read_link (Lee un eslabón)

Precondiciones:

No hay.

Poscondiciones:

No hay.

Entradas:

lr	El apuntador al eslabón que contiene la dirección del eslabón a leer

Salidas:

lr	Registro eslabón elido

Proceso:

Si la dirección del eslabón a leer es nula
    Error
Obtén el buffer indicado en la dirección del eslabón
Mueve el registro del bloque de datos al área del registro usando el offset de la dirección
Libera el bloque de datos

7.6.18 Set::update_link (Actualiza un eslabón)

Precondiciones:

No hay.

Poscondiciones:

No hay.

Entradas:

lr	El apuntador al eslabón que contiene la dirección del eslabón a leer

Salidas:

lr	Registro eslabón leído

Proceso:

Si la dirección del eslabón a leer es nula
    Error
Obtén el buffer indicado en la dirección del eslabón
Mueve el registro al bloque de datos usando el offset de la dirección
Marca el bloque como modificado

7.7 Diseño funcional de la clase Cache

En el sistema, todas las operaciones básicas de entrada y salida están controladas por un objeto de la clase **Cache**. Las operaciones básicas son:

- La obtención de un nuevo bloque
- La escritura de un bloque
- La lectura de un bloque
- La liberación de un bloque

El cache mantiene un mecanismo para minimizar los accesos a disco mediante el cual los bloque más recientemente utilizados se mantienen en buffers de memoria para atender posibles próximas solicitudes. Al requerirse un nuevo buffer, se le asigna al usuario aquel que menos recientemente se ha utilizado.

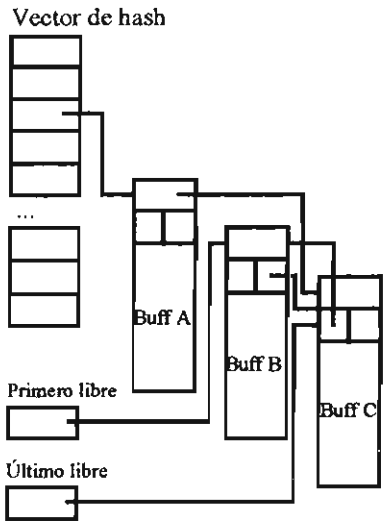
El cache de buffers está constituido por un vector de hash que contiene apuntadores a los buffers que representan un bloque del disco o que se solicitaron para tal propósito. La entrada en el vector de hash para un buffer, se determina con la siguiente fórmula:

entrada = (fileh + blknr) % número de entradas del hash

en donde fileh es el asa del archivo y blknr el número de bloque.

Además del vector de hash, existe una lista de buffers libres. La lista contiene los buffers que pueden ser reasignados, y está implementada como una lista doblemente ligada. Cada vez que se libera un buffer se agrega al final de la lista , y cada vez que se requiere un buffer, se toma del principio de la lista. De esta manera los buffers asignados serán los menos recientemente utilizados.

El siguiente diagrama muestra esquemáticamente la estructura del cache de buffers.



El diagrama nos muestra que, en una entrada del vector de hash, se encuentran los buffers que corresponden a los bloques A, B y C. Los buffers de los bloques B y C se encuentran en la lista de buffers disponibles. Inicialmente todos los buffers se encuentran en la lista de buffers disponibles y todas las listas del vector de hash estarán vacías. A medida que se reciben solicitudes de buffers, éstos se incorporan en el hash que les corresponde y se eliminan de la lista de buffers libres. Cuando un buffer se libera, se incorpora a la lista de buffers libres.

En general, un buffer puede estar en uno de los siguientes estados:

a). Se encuentra solamente en la lista de buffers libres. En este caso no contiene información válida (no refleja el estado de ningún bloque del disco). Cuando se requiere de un nuevo buffer se toma del principio de la lista de buffers libres.

b). Se encuentra solamente en el hash. En este caso está siendo utilizado por el sistema. Ya sea que su contenido se haya leído del disco o que se haya solicitado un nuevo buffer para crear un nuevo bloque de disco.

c). Se encuentra tanto en el hash como en la lista de buffers libres. Esto nos indica que el buffer contienen información válida de un bloque de disco pero que no está siendo usando por ahora. Los buffers en este estado pueden ser solicitados nuevamente para acceder a la información que contienen o pueden ser usados para contener información de un bloque diferente al que contienen, en este último caso, será necesario escribir en disco su contenido antes de reutilizarlo.

Los objetos de la clase Cache contienen la siguiente información:

buf_hash. Vector de apuntadores a estructuras buf. Cada apuntador tiene la dirección del primer buffer de una cadena de buffers.

free_list_head. Apuntador al primer buffer de la cadena de buffers disponibles.

free_list_tail. Apuntador al último buffer de la cadena de buffers disponibles.

Además, la clase utiliza la estructura buf la cual contiene los siguientes elementos de información:

fileh. Asa del archivo al que está asignado el buffer.

blknr. Número del bloque para el cual se asignó el buffer.

status. Contiene la suma lógica de los siguientes valores:

BUFF_STATUS_VAL. Indica que el buffer contiene información válida de un bloque del disco.

BUFF_STATUS_MOD. Indica que el contenido del buffer se ha modificado, por lo tanto antes de reasignarlo a otro bloque será necesario escribirlo en el disco.

prev. Apuntador al buffer anterior en la lista de buffers libres.

next. Apuntador al siguiente buffer en la lista de buffers libres.

hash. Apuntador al siguiente buffer en una lista del hash de buffers.

A continuación el buffer contiene una unión que se usa para describir los diferentes tipos de bloques que se manejan en el sistema.

Las funciones públicas del cache son las siguientes:

7.7.1 Cache::Cache (Constructor)

La inicialización del cache de buffers consiste en obtener la memoria para un determinado número de buffers y formar la lista de buffers libres. Inicialmente todas las entradas del hash estarán vacías.

Precondiciones:

No hay

Poscondiciones:

Todos los bufferes se encuentran en la lista de buffers libres.

Todas las entradas del hash tienen valores nulos.

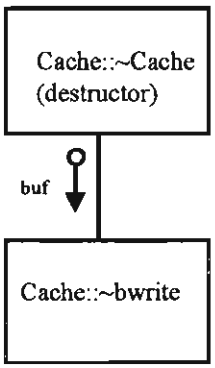
Proceso:

Para i = 1 a dimensión del hash
    Asigna hash[i] = nulo
Asigna nulo a la cabeza de la lista de buffers libres
Asigna nulo a la cola de la lista de buffers libres
Repite número_buffers veces
    Solicita memoria para un buffer
    Agrega el buffer al principio de la lista de buffers libres

7.7.2 Cache::~Cache (Destructor)

Al dar por terminada la existencia del cache se revisan todos los buffers y se graban en disco todos aquellos que han sido modificado y cuya información aún no se refleja en el disco.

Diagrama de estructura:



Precondiciones:

El cache debe estar inicializado y posiblemente utilizado

Poscondiciones:

Desaparece el cache y con sus servicios.

Proceso:

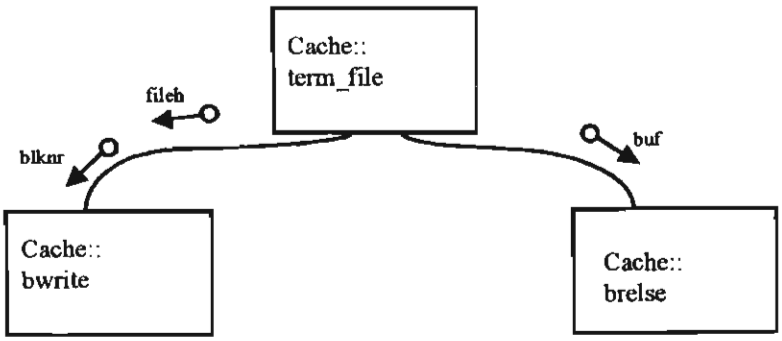
Por cada una de las entradas del vector de hash
    Por cada uno de los buffers del vector
        Si el buffer se ha modificado (su contenido es diferente de cuando se leyó)
            Graba el buffer en disco

Por cada buffer en la lista de buffers libres
Si el buffer se ha modificado (su contenido es diferente de cuando se leyó)
Graba el buffer en disco

7.7.3 Cache::term_file (Da por terminado el proceso de un archivo)

Esta función se invoca cuando se cierra un archivo y tiene como propósito grabar en disco todos los bloques que hayan quedado pendientes en el cache y liberarlos para ser usados en otros archivos.

Diagrama de estructura:



Precondiciones:

No hay

Poscondiciones:

Todos los bloques del archivo quedan grabados en disco.

Entradas:

Fileh El asa del archivo

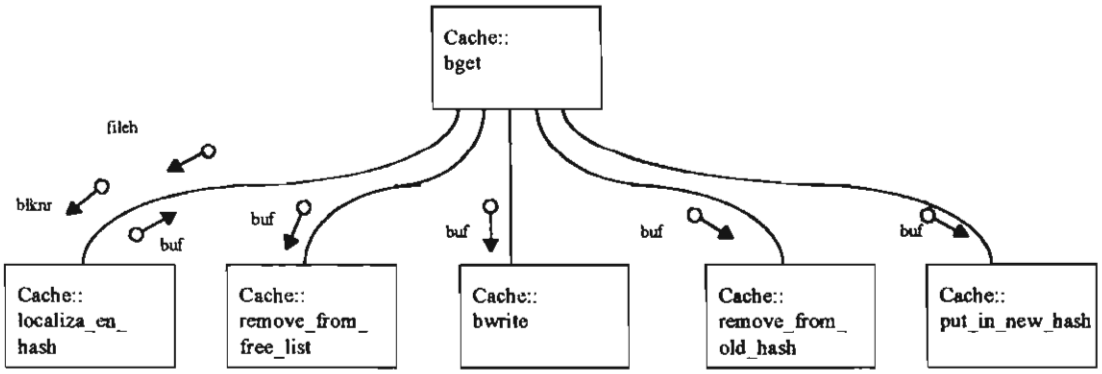
Proceso:

Por cada una de las entradas del vector de hash
Por cada uno de los buffers del vector
Si el buffer pertenece al archivo y ha sido modificado
Graba el bloque en disco
Elimina el bloque del hash
Si no se encuentra en la lista de buffers libres
Libera el bloque

7.7.4 Cache::bget (Proporciona un nuevo buffer)

La función proporciona un buffer para contener un bloque de un archivo.

Diagrama de estructura:



Precondiciones:

El bloque está en hash o existe algún bloque libre

Poscondiciones:

El bloque queda en hash
El bloque se elimina de la lista de libres

Entradas:

fileh	Asa del archivo para el cual se hace la solicitud
blknr	Número de bloque dentro del archivo

Salidas:

buf	Apuntador al buffer

Proceso:

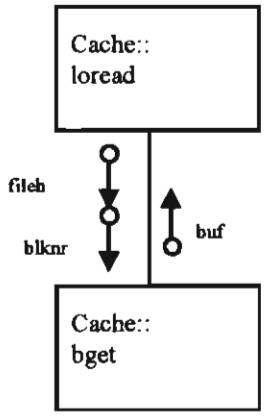
```
Si el bloque está en hash
        Elimínalo de lista de bloques libres
        Regresa el bloque
sino
        Toma primer bloque de lista de libres
        Si fue modificado
                Grábalo en disco
        Elimínalo de lista de bloques libres
```

Elimínalo de su hash anterior
Inserta bloque en su nuevo hash
Regresa el bloque

7.7.5 Cache::bread (Lee un bloque de disco)

Primeramente se trata de satisfacer la solicitud con la información contenida en el cache. Solamente que el bloque no se encuentre en el cache se lleva a cabo una lectura del disco.

Diagrama de estructura:



Precondiciones:

El archivo debe encontrarse abierto.
El número de bloque debe ser válido

Poscondiciones:

El bloque se elimina de la lista de bloques libres.
El bloque queda en el hash marcado como válido.

Salidas:

buf	El apuntador al buffer leído

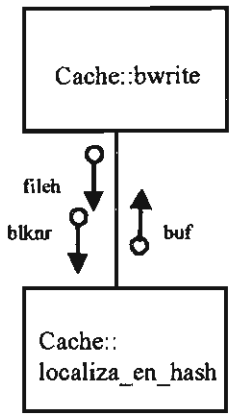
Funciones:

Obtiene el bloque con get_block
Si contiene información válida
    Regresa el buffer
Lee bloque de disco
Marca el estado del bloque como válido
Regresa el bloque

7.7.6 Cache::bwrite (Graba un bloque físicamente en el disco)

Esta es una función privada, ya que solamente otras funciones miembros de Cache la invocan.

Diagrama de estructura:



Precondiciones:

El buffer debe encontrarse en el hash

Poscondiciones:

El bloque queda marcado como no modificado.

Entradas:

buf Apuntador al buffer que se va a escribir

Salidas:

No hay

Proceso:

Si el buffer no se encuentra en el hash
    Error
Si el buffer ha sido modificado
    Graba el bloque en el disco
    Elimina la marca de modificado en el buffer

7.7.7 Cache::brelse (libera un buffer)

El buffer se libera. El bloque no se graba en disco, sino simplemente se agrega al final de la lista de buffers libres de donde eventualmente puede ser tomado para su reasignación.

Precondiciones:

El bloque no debe estar en la lista de bloques libres.

Poscondiciones:

El bloque queda en la lista de buffers libres.

Entradas:

buf	Apuntador al buffer

Salidas:

No hay

Proceso:

```
Si el buffer está en la lista de buffers libres
        Error
Coloca el buffer al final de la lista de buffers libres
```

7.7.8 Cache::localiza_en_hash (Localiza un buffer en el hash)

Precondiciones:

No hay

Poscondiciones:

No hay

Entradas:

fileh	Asa del archivo para el que se usa el buffer.
blknr	Número de bloque dentro del archivo.

Salidas:

buf Apuntador al buffer en cuestión.

Proceso:

Calcula el índice del vector de hash
Busca secuencialmente en la lista de buffer
Si se encuentra
    Regresa el apuntador al buffer
sino
    Regresa nulo

7.7.9 Cache::remove_from_free_list (Elimina un buffer de la lista de buffers libres)

Al eliminar el buffer de la lista de buffers libres, este no puede ser utilizado para contener ningún otro bloque de información. Para que pueda volver a ser usado tendrá que invocarse a la función Cache::brelse o Cache::bmodify.

Precondiciones:

El buffer debe encontrarse en la lista de buffers libres.

Poscondiciones:

El buffer ya no se encuentra en la lista de buffers libres.

Entradas:

buf Apuntador el buffer.

Salidas:

No hay.

Proceso:

Si hay un buffer anterior en la lista
    Modifica el apuntador al siguiente buffer del buffer anterior
sino
    Modifica el apuntador al primer buffer de la lista.
Si hay un buffer siguiente
    Modifica el apuntador al anterior buffer del buffer siguiente

sino

Modifica el apuntador al último buffer de la lista.

7.7.10 Cache::remove_from_old_hash (Elimina un buffer del hash)

Esta función se invoca cuando un buffer que contiene información de un bloque que no ha sido modificado o que ya se grabó en el disco va a ser utilizado para contener información de otro bloque.

Precondiciones:

El buffer se encuentra en el hash.

Poscondiciones:

El buffer ya no se encuentra en el hash.

Entradas:

buf El apuntador al buffer

Salidas:

No hay

Proceso:

Calcula la entrada que le corresponde en el vector de hash
Localiza el buffer a eliminar (búsqueda secuencial)
Elimina el buffer de la lista

7.7.11 Cache::put_in_new_hash (Coloca un buffer en el hash)

Esta función se invoca cuando se asigna un buffer a un bloque de un archivo. Al colocarlo en el hash se podrá localizar fácilmente cuando se requiera.

Precondiciones:

El buffer no debe encontrarse en el hash.

Poscondiciones:

El buffer se encuentra en el hash en la lista que le corresponde.

Entradas:

buf Apuntador al buffer que se va a incorporar al hash.

Salidas:

No hay.

Proceso:

Calcula la entrada que le corresponde en el vector del hash.
Inserta el buffer al principio de la lista.

7.7.12 Cache::bmodify (Marca un buffer como modificado)

Marca el buffer como modificado y lo libera. El bloque no se graba en disco, sino simplemente se agrega al final de la lista de buffers libres de donde eventualmente puede ser tomado para su reasignación. Antes de reasignarlo para contener la información de otro bloque, será necesario grabar su contenido en el disco.

Precondiciones:

El bloque no debe estar en la lista de bloques libres.

Poscondiciones:

El bloque queda en la lista de buffers libres.

Entradas:

buf Apuntador al buffer

Salidas:

No hay

Proceso:

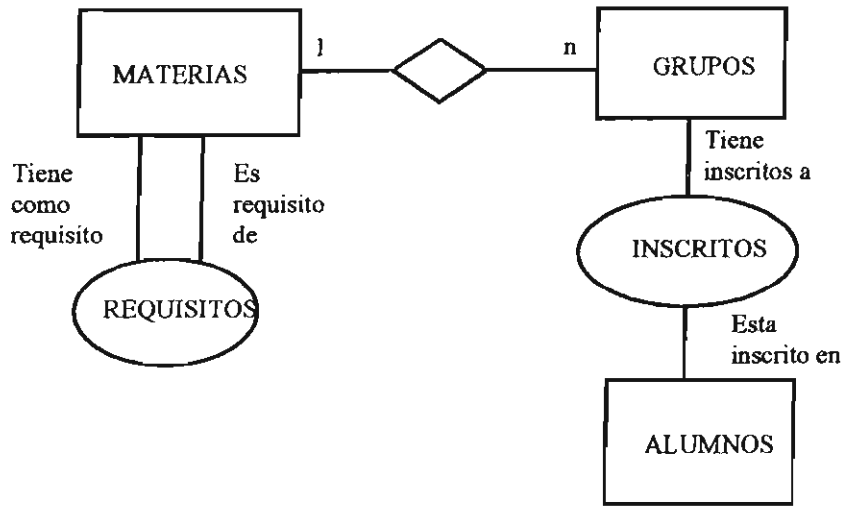
Si el buffer está en la lista de buffers libres
    Error
Coloca el buffer al final de la lista de buffers libres

8. Pruebas del sistema

Para la prueba del sistema se elaboraron un conjunto de programas con los que se intenta probar todas las funciones y sobre todo crear condiciones para que las funciones respondan ante una variedad de casos. Naturalmente, el conjunto de pruebas no nos puede asegurar que no existan errores. A continuación se describe cada programa de prueba.

8.1 demo1

El programa demo1 crea tres archivos y dos sets de un sistema escolar. Los archivos son: materias, alumnos y grupos. Los sets son inscritos y requisitos. El diagrama de la base de datos es el siguiente:



En el programa se realizan las siguientes pruebas:

1). Creación de un objeto Database vacío en el archivo ESCOLAR.DB.

2). Creación y carga de los tres archivos con un número limitado de datos. Dado que los tres archivos son objetos File_db se crea un índice con un segmento tipo serial en cada uno de ellos. Además, en el archivo de materias se crea un índice con la clave de la materia. En el archivo de alumnos se crean dos índices, uno con la matrícula y el otro con el nombre. En el archivo de grupos se crea una llave concatenada con la clave del grupo y la clave de la materia.

3). Creación de los dos sets: inscritos y requisitos.

4). Cada uno de los archivos se lista para verificar su correcto contenido. El archivo de alumnos se lista en los siguientes órdenes: por matrícula en orden ascendente,

por matrícula en orden descendente, por nombre, por nombre a partir de cierta llave y por orden físico.

5). Eliminación de registros del archivo de alumnos.

6). Repetición de los listados para asegurar que las bajas de alumnos dejaron una estructura consistente en los archivos.

7). Eliminación de algunos miembros del set de inscritos.

8). Obtención de las listas de grupo y tiras de materias. Esta prueba asegura la integridad de la estructura de un set después de la eliminación de algunos de los miembros.

Este programa no requiere de ningún prerrequisito y se puede correr todas las veces que sea necesario sin necesidad de inicialización ni borrado de archivos. El programa elimina y crea los archivos que utiliza.

El programa tiene una pequeña función independiente para realizar cada una de las operaciones por lo que resulta sencillo modificarlo y agregarle nuevas opciones de prueba.

A continuación se presenta el listado del programa:

```
1  /*
   2        Programa: DEMO1
   3        Funciones:
   4              Elimina la base de datos escolar.db
   5              Crea la base de datos escolar.db
   6              Crea los archivos:
   7                    materias.dat
   8                    alumnos.dat
   9                    grupos.dat
  10              Crea los sets
  11                    inscritos.dat
  12                    requisitos.dat
  13              Lista el contenido de los archivos
  14              Elimina algunos alumnos
  15              Elimina algunos miembros de los sets
  16              Lista nuevamente el contenido de los archivos y sets
  17
  18 */
  19 #include <iostream.h>
  20 #include <string.h>
  21 #include <stdio.h>
  22 #include <stdlib.h>
  23 #include "database.h"
  24 #define MAX_VPT 10
  25 #define MAX_RC 100
  26
  27
```

```
28  //    Estructura de los registros
29
30  typedef struct {
31        Serial      ser;
32        int   cve;
33        char  nom[ 31] ;
34        int   cred;
35  } Reg_materia;
36
37  typedef     struct {
38        Serial      ser;
39        int   mat;
40        char  nom[ 26] ;
41  } Reg_alumno;
42
43  typedef struct {
44        Serial      ser;
45        int   cve;
46        int   mat_cve;
47  } Reg_grupo;
48
49  typedef struct {
50        int   gpo_cve;
51        int   mat_cve;
52        int   alm_mat;
53  } Reg_inscrito;
54
55
56  //    Funciones para listar los registros de datos
57
58
59  ostream &operator << (ostream &s, Reg_materia &r)
60  {
61
62        return s<<r.ser<<"."<<r.cve<<"."<<r.nom<<"."<<r.cred;
63  }
64  ostream &operator << (ostream &s, Reg_alumno &r)
65  {
66
67        return s << r.ser << "." << r.mat << "."  << r.nom;
68  }
69  ostream &operator << (ostream &s, Reg_grupo &r)
70  {
71
72        return s << r.ser << "." << r.cve << "."  << r.mat_cve;
73  }
74
75  //    Declaración de fucniones
76
77  File_db     *creacion_materias      (Database *db);
78  void  lista_materias          (File_db *arch_materias);
79  File_db *creacion_alumnos     (Database *db);
80  void  lista_alumnos_matricula       (File_db *arch_alumnos);
81  void  lista_alumnos_inverso   (File_db *arch_alumnos);
82  void  lista_alumnos_nombre    (File_db *arch_alumnos);
83  void  lista_alumnos_nombre_a_partir_de(File_db  *arch_alumnos,  char
*inicial);
84  void  lista_alumnos_orden_fisico(File_db *arch_alumnos);
85  File_db *creacion_grupos      (Database *db);
```

```
  86  void  lista_grupos            (File_db *arch_grupos);
  87  Set   *creacion_inscritos     (Database *db, File_db *arch_alumnos,
  88                                 File_db *arch_grupos);
  89  void  lista_de_grupo          (Set      *set_inscritos,      File_db
*arch_grupos);
  90  void  tiras_de_materias       (Set      *set_inscritos,      File_db
*arch_alumnos, File_db *arch_materias);
  91  void  baja_de_alumnos         (File_db *arch_alumnos);
  92  Set   *creacion_requisitos    (Database        *db,         File_db
*arch_materias);
  93  void  lista_requisitos  (Set        *set_requisitos,        File_db
*arch_materias);
  94  void  lista_requisitos_de     (Set      *set_requisitos,     File_db
*arch_materias);
  95  void  elimina_alumnos_de_grupo(Set      *set_inscritos,      File_db
*arch_grupos,
  96                           File_db *arch_alumnos);
  97  void  prueba_navegacion (File_db *arch_alumnos);
  98
  99  //    Programa Principal
 100
 101  void main (int argc, char *argv[])
 102  {
 103
 104        remove("ESCOLAR.DB");
 105          Database db("ESCOLAR.DB");
 106
 107        File_db *arch_materias, *arch_alumnos, *arch_grupos;
 108        Set *set_inscritos, *set_requisitos;
 109
 110        arch_materias     = creacion_materias(&db);
 111        arch_alumnos      = creacion_alumnos(&db);
 112        arch_grupos = creacion_grupos(&db);
 113
 114        set_inscritos     =   creacion_inscritos(&db,   arch_alumnos,
arch_grupos);
 115        set_requisitos = creacion_requisitos      (&db,
arch_materias);
 116
 117        lista_materias    (arch_materias);
 118
 119        lista_alumnos_matricula (arch_alumnos);
 120        lista_alumnos_inverso(arch_alumnos);
 121        lista_alumnos_nombre (arch_alumnos);
 122        lista_alumnos_nombre_a_partir_de(arch_alumnos, "Contreras");
 123        lista_alumnos_orden_fisico(arch_alumnos);
 124
 125        lista_grupos      (arch_grupos);
 126        lista_de_grupo    (set_inscritos, arch_grupos);
 127        tiras_de_materias(set_inscritos,                arch_alumnos,
arch_materias);
 128
 129        baja_de_alumnos   (arch_alumnos);
 130
 131        lista_alumnos_matricula (arch_alumnos);
 132        lista_alumnos_nombre (arch_alumnos);
 133        lista_alumnos_nombre_a_partir_de(arch_alumnos, "Contreras");
 134        lista_alumnos_orden_fisico(arch_alumnos);
 135        lista_de_grupo    (set_inscritos, arch_grupos);
```

```
136         tiras_de_materias(set_inscritos,                    arch_alumnos,
arch_materias);
137         lista_requisitos(set_requisitos, arch_materias);
138         lista_requisitos_de(set_requisitos, arch_materias);
139
140         elimina_alumnos_de_grupo(set_inscritos,
arch_grupos,arch_alumnos);
141
142         lista_de_grupo    (set_inscritos, arch_grupos);
143         tiras_de_materias(set_inscritos,                    arch_alumnos,
arch_materias);
144         prueba_navegacion(arch_alumnos);
145           arch_alumnos->stat();
146         cout << "Terminación normal del programa\n";
147
148   }
149
150
151   //    Creación y carga del archivo de arch_materias
152   File_db *creacion_materias(Database *db)
153   {
154         static Reg_materia      alta_materias[] = {
155               { 0, 101, "Cálculo Diferencial", 12},
156               { 0, 102, "Calculo Integral",  10},
157               { 0, 103, "Algebra Lineal",    10},
158               { 0, 104, "Ecuaciones Diferenciales", 9},
159               { 0, 105, "Probabilidad y Estadística", 10},
160               { 0, 106, "Investigación de Operaciones", 10},
161               { 0, 201, "Computación I", 9},
162               { 0, 202, "Computación II", 9},
163               { 0, 203, "Programación Avanzada", 9},
164               { 0, 301, "Circuitos Lógicos I", 9},
165               { 0, 302, "Circuitos Lógicos II", 10},
166               { 0, 303, "Arquitectura de Computadoras I", 10},
167               { 0, 303, "Comunicaciones I", 10},
168               { 0, 304, "Procesamiento de Señales", 10},
169               { 0, 0}
170         };
171         Reg_materia *p_materia, reg_materia;
172
173         struct idx_seg Seg_materia_cve[] = {{
174               (char*)&reg_materia.cve-(char*)&reg_materia.ser,
175               sizeof(reg_materia.cve), SEG_TYPE_INT, 0}};
176           File_db *arch_materias;
177
178         cout << "* * * Creación y Carga de arch_materias * * *\n\n";
179         remove("MATERIAS.DAT");
180           arch_materias = db->get_file("MATERIAS.DAT");
181         arch_materias->createx("clave", 1, Seg_materia_cve);
182         for (p_materia = alta_materias; p_materia->cve; p_materia++)
183               arch_materias->write(p_materia, sizeof (*p_materia));
184         return arch_materias;
185
186   }
187
188
189   //    Listado del archivo de arch_materias
190   void lista_materias(File_db *arch_materias)
191   {
```

```
192         Reg_materia reg_materia;
193
194         cout << "\n* * * Listado del archivo de Materias * * *\n";
195         arch_materias->index("clave");
196         reg_materia.cve = 0;
197         while(arch_materias->read_gt(&reg_materia))
198               cout << reg_materia << "\n";
199   }
200
201   //    Creación y carga del archivo de arch_alumnos
202   File_db *creacion_alumnos(Database *db)
203   {
204         static Reg_alumno alta_alumnos[] = {
205               {0, 9701, "Fajardo Victor"},
206               {0, 9702, "Flores Alberto"},
207               {0, 9703, "Galindo Raymundo"},
208               {0, 9704, "Zamudio Alonso"},
209               {0, 9705, "Huerta Carlos"},
210               {0, 9706, "Alvarez Martin"},
211               {0, 9707, "Bautista Luis"},
212               {0, 9708, "Barrera Antonio"},
213               {0, 9709, "Contreras Francisco"},
214               {0, 9710, "Manrique Jorge"},
215               {0, 9711, "Pacheco Javier"},
216               {0, 9712, "Plascencia Raúl"},
217               {0, 9713, "Herrera Laura"},
218               {0, 9714, "Dorantes Armando"},
219               {0, 9715, "Elordui Marta"},
220               {0, 9716, "Fernandez Dolores"},
221               {0, 0}
222         };
223         File_db *arch_alumnos;
224
225         remove("ALUMNOS.DAT");
226         Reg_alumno *p_alumno, reg_alumno;
227         struct idx_seg Seg_alumnos_mat[] = {{
228               (char*)&reg_alumno.mat-(char*)&reg_alumno.ser,
229               sizeof(reg_alumno.mat), SEG_TYPE_INT, 0}};
230         struct idx_seg Seg_alumnos_nom[] = {{
231               (char*)reg_alumno.nom-(char*)&reg_alumno.ser,
232               sizeof(reg_alumno.nom), SEG_TYPE_CHAR, 0}};
233
234         cout << "* * * Carga de arch_alumnos * * *\n\n";
235           arch_alumnos = db->get_file("ALUMNOS.DAT");
236         arch_alumnos->createx("matricula", 1, Seg_alumnos_mat);
237         arch_alumnos->createx("nombre", 1, Seg_alumnos_nom);
238         for (p_alumno = alta_alumnos; p_alumno->mat; p_alumno++)
239               arch_alumnos->write(p_alumno, sizeof (*p_alumno));
240
241         return arch_alumnos;
242   }
243
244
245
246   //    Listado del archivo de alumnos por matrícula
247   void lista_alumnos_matricula(File_db *arch_alumnos)
248   {
249         Reg_alumno reg_alumno;
250         arch_alumnos->index("matricula");
```

```
 251         reg_alumno.mat = 0;
 252
 253         cout << "\n* * * Listado del archivo de alumnos por matrícula
* * *\n";
 254         while(arch_alumnos->read_gt(&reg_alumno))
 255               cout << reg_alumno << "\n";
 256   }
 257
 258   //    Listado del archivo de alumnos por matrícula en orden inverso
 259   void lista_alumnos_inverso(File_db *arch_alumnos)
 260   {
 261         Reg_alumno reg_alumno;
 262         arch_alumnos->index("matricula");
 263         reg_alumno.mat = 9999;
 264
 265         cout << "\n* * * Listado del archivo de alumnos por matrícula
en orden inverso * * *\n";
 266         while(arch_alumnos->read_lt(&reg_alumno))
 267               cout << reg_alumno << "\n";
 268   }
 269   //    Listado del archivo de alumnos por nombre
 270   void lista_alumnos_nombre(File_db *arch_alumnos)
 271   {
 272         Reg_alumno reg_alumno;
 273         arch_alumnos->index("nombre");
 274         reg_alumno.nom[ 0] = 0;
 275
 276         cout << "\n* * * Listado del archivo de alumnos por nombre *
* *\n";
 277         while(arch_alumnos->read_gt(&reg_alumno))
 278               cout << reg_alumno << "\n";
 279   }
 280   //    Listado del archivo de alumnos por nombre a partir de una
llave
 281    void lista_alumnos_nombre_a_partir_de(File_db *arch_alumnos, char
*inicial)
 282   {
 283         Reg_alumno reg_alumno;
 284         arch_alumnos->index("nombre");
 285         strcpy (reg_alumno.nom, "Contreras");
 286
 287         cout << "\n* * * Listado del archivo de alumnos por nombre a
partir de " << inicial << "\n";
 288         while(arch_alumnos->read_gt(&reg_alumno))
 289               cout << reg_alumno << "\n";
 290   }
 291   //    Listado del archivo de alumnos por orden físico
 292   void lista_alumnos_orden_fisico(File_db *arch_alumnos)
 293   {
 294         Reg_alumno reg_alumno;
 295         arch_alumnos->start();
 296
 297         cout << "\n* * * Listado del archivo de alumnos por orden
fisico * * *\n";
 298         while(arch_alumnos->next(&reg_alumno) != EOF)
 299               cout << reg_alumno << "\n";
 300   }
 301
 302   //    Creación de Grupos
```

```
303  File_db *creacion_grupos(Database *db)
304  {
305        Reg_grupo   alta_grupos[ ] = {
306              { 0, 1001, 101},
307              { 0, 1002, 101},
308              { 0, 1101, 102},
309              { 0, 1102, 102},
310              { 0, 1103, 102},
311              { 0, 1201, 201},
312              { 0, 1201, 202},
313              { 0, 1201, 203},
314              { 0,0,0}
315        };
316          File_db *arch_grupos;
317        Reg_grupo *p_grupo, reg_grupo;
318        struct idx_seg Seg_grupos [ ] = {
319              { (char *)&reg_grupo.cve - (char *)&reg_grupo.ser,
320                          sizeof(reg_grupo.cve), SEG_TYPE_INT, 0},
321              { (char *)&reg_grupo.mat_cve - (char *)&reg_grupo.ser,
322                          sizeof(reg_grupo.mat_cve), SEG_TYPE_INT, 0}
323        };
324
325
326          cout << "* * * Carga de arch_grupos * * *\n\n";
327        remove("GRUPOS.DAT");
328          arch_grupos = db->get_file("GRUPOS.DAT");
329        arch_grupos->createx("gpo_mat", 2, Seg_grupos);
330          for (p_grupo = alta_grupos; p_grupo->cve; p_grupo++)
331              arch_grupos->write(p_grupo, sizeof(*p_grupo));
332          return arch_grupos;
333  }
334
335  //    Listado de arch_grupos
336  void lista_grupos(File_db *arch_grupos)
337  {
338        Reg_grupo reg_grupo;
339        cout << "\n* * * Listado del archivo de grupos * * *\n";
340        arch_grupos->index("SERIAL");
341        reg_grupo.ser = 0;
342        while(arch_grupos->read_gt(&reg_grupo))
343              cout << reg_grupo << "\n";
344  }
345
346
347
348
349  //    creación del Set de Inscritos
350  Set *creacion_inscritos(Database *db, File_db *arch_alumnos,
351                    File_db *arch_grupos)
352  {
353        Reg_inscrito      alta_inscritos[ ] = {
354              { 1001, 101, 9701},
355              { 1001, 101, 9702},
356              { 1001, 101, 9705},
357              { 1002, 101, 9703},
358              { 1002, 101, 9704},
359              { 1101, 102, 9701},
360              { 1101, 102, 9702},
361              { 1101, 102, 9705},
```

```
362                 {1102, 102, 9703},
363                 {1102, 102, 9704},
364                 {1201, 201, 9706},
365                 {1201, 201, 9707},
366                 {1201, 201, 9708},
367                 {1201, 201, 9709},
368                 {1201, 201, 9710},
369                 {1201, 201, 9711},
370                 {1201, 201, 9712},
371                 {1201, 201, 9713},
372                 {1201, 201, 9714},
373                 {1201, 201, 9715},
374                 {1201, 201, 9716},
375                 {0, 0, 0}
376         };
377         Set *set_inscritos;
378         Reg_inscrito *p_inscrito;
379         Reg_grupo reg_grupo;
380         Reg_alumno reg_alumno;
381         remove("INSCRITO.SET");
382
383         set_inscritos   =   db->get_set("INSCRITO.SET",   arch_grupos,
arch_alumnos);
384
385         cout << "* * * Carga del set de set_inscritos * * *\n\n";
386         arch_grupos->index("gpo_mat");
387         arch_alumnos->index("matricula");
388         for(p_inscrito    =    alta_inscritos;    p_inscrito->gpo_cve;
p_inscrito++) {
389                 reg_alumno.mat = p_inscrito->alm_mat;
390                 if (!arch_alumnos->read_eq(&reg_alumno)) {
391                         cout << "  No se pudo leer al reg_alumno: " <<
reg_alumno;
392                         break;
393                 }
394                 reg_grupo.cve = p_inscrito->gpo_cve;
395                 reg_grupo.mat_cve = p_inscrito->mat_cve;
396                 if (!arch_grupos->read_eq(&reg_grupo)) {
397                         cout << "  No se pudo leer el reg_grupo: " <<
reg_grupo;
398                         break;
399                 }
400                 set_inscritos->add_member_end(reg_grupo.ser,
reg_alumno.ser);
401         }
402         return set_inscritos;
403 }
404
405
406
407 //     Impresión de listas de grupos
408 void lista_de_grupo(Set *set_inscritos, File_db *arch_grupos)
409 {
410         Reg_grupo reg_grupo;
411         Reg_alumno reg_alumno;
412
413         cout << "\n* * * Impresión de listas de grupo * * *\n";
414         arch_grupos->index("SERIAL");
415         reg_grupo.ser = 0;
```

```
416         while(arch_grupos->read_gt(&reg_grupo)) {
417               cout << "\nListado del reg_grupo:" << reg_grupo <<
"\n";
418               if (set_inscritos->get_first_member(&reg_alumno))
419                     do
420                           cout << "\t" << reg_alumno << "\n";
421                     while(set_inscritos-
>get_next_member(&reg_alumno));
422               cout << "\n";
423         }
424   }
425
426   //    Impresión de las tiras de materias
427    void tiras_de_materias(Set *set_inscritos, File_db *arch_alumnos,
File_db *arch_materias)
428   {
429         Reg_grupo reg_grupo;
430         Reg_alumno reg_alumno;
431         Reg_materia reg_materia;
432
433         cout << "\n* * * Impresión de tiras de UEA's * * *\n";
434
435         arch_materias->index("clave");
436         arch_alumnos->index("SERIAL");
437         reg_alumno.ser = 0;
438         while(arch_alumnos->read_gt(&reg_alumno)) {
439               cout << "Tira del reg_alumno: " << reg_alumno << "\n";
440               if (set_inscritos->get_first_owner(&reg_grupo)) {
441                     do {
442                           reg_materia.cve = reg_grupo.mat_cve;
443                           arch_materias->read_eq(&reg_materia);
444                           cout << "\t" << reg_grupo << "-" <<
reg_materia <<"\n";
445                     }
446                     while(set_inscritos->get_next_owner(&reg_grupo));
447               }
448         }
449
450   }
451
452
453   //    Baja de algunos arch_alumnos
454   void baja_de_alumnos(File_db *arch_alumnos)
455   {
456         Reg_alumno  baja_alumnos[ ] = {
457               { 0, 9702, "Flores Alberto"},
458               { 0, 9712, "Plascencia Raúl"},
459               { 0, 9713, "Herrera Laura"},
460               { 0, 9714, "Dorantes Armando"},
461               { 0, 0}
462         };
463         cout << "\n* * * Se dan de baja los siguientes alumnos * *
*\n";
464         Reg_alumno *p_alumno, reg_alumno;
465
466           arch_alumnos->index("matricula");
467         for (p_alumno = baja_alumnos; p_alumno->mat; p_alumno++) {
468               reg_alumno = *p_alumno;
469               if(!arch_alumnos->read_eq(&reg_alumno))
```

```
470                     fatal_error("No se leyó el registro a eliminar");
471               cout <<  "\t" << reg_alumno << "\n";
472               arch_alumnos->del();
473         }
474   }
475
476   //    Creación del set de requisitos
477   Set *creacion_requisitos(Database *db, File_db *arch_materias)
478   {
479
480         struct {
481               int mat;
482               int req;
483         } requisitos[] = {
484               {102, 101},
485               {104, 102},
486               {104, 103},
487               {106, 105},
488               {106, 104},
489               {202, 201},
490               {203, 202},
491               {203, 103},
492               {302, 301},
493               {303, 302},
494               {303, 203},
495               {0,0}
496         }, *p_req;
497         Set *set_requisitos;
498         Reg_materia reg_materia, reg_requisito;
499         cout << "* * * Creación Set de requisitos * * *\n";
500         remove("REQUIS.SET");
501         set_requisitos = db->get_set("REQUIS.SET", arch_materias,
502                                     arch_materias);
503         arch_materias->index("clave");
504         for (p_req = requisitos; p_req->mat; p_req++) {
505               reg_materia.cve = p_req->mat;
506               if (!arch_materias->read_eq(&reg_materia)) {
507                     cout << "Materia no encontrada:" << reg_materia;
508                     continue;
509               }
510               reg_requisito.cve = p_req->req;
511               if (!arch_materias->read_eq(&reg_requisito)) {
512                     cout    <<    "Requisito    no    encontrado:"    <<
reg_materia;
513                     continue;
514               }
515               set_requisitos->add_member_end(reg_materia.ser,
reg_requisito.ser);
516         }
517         return set_requisitos;
518   }
519
520
521   //    Listado de los requisitos de una materia
522   void lista_requisitos(Set *set_requisitos, File_db *arch_materias)
523   {
524         Reg_materia reg_materia, reg_requisito;
525
```

```
526         cout << "\n* * * Listado de Materias y sus requisitos * *
*\n";
527         arch_materias->index("SERIAL");
528         reg_materia.ser = 0;
529         while(arch_materias->read_gt(&reg_materia)) {
530               cout << reg_materia << "\n";
531               if (set_requisitos->get_first_member(&reg_requisito))
532                     do
533                           cout << "\t" << reg_requisito << "\n";
534               while(set_requisitos->get_next_member(&reg_requisito));
535         }
536   }
537
538   //    Listado de las materias que dependen de un requisito
539         void    lista_requisitos_de(Set    *set_requisitos,    File_db
*arch_materias)
540   {
541         Reg_materia reg_materia, reg_requisito;
542
543         cout << "\n* * * Listado de Materias y aquellas que las
requeren * * *\n";
544         arch_materias->index("SERIAL");
545         reg_materia.ser = 0;
546         while(arch_materias->read_gt(&reg_materia)) {
547               cout << reg_materia << "\n";
548               if (set_requisitos->get_first_owner(&reg_requisito))
549                     do
550                           cout << "\t" << reg_requisito << "\n";
551               while(set_requisitos->get_next_owner(&reg_requisito));
552         }
553   }
554
555   //    Eliminación de miembros del set de inscritos
556       void   elimina_alumnos_de_grupo(Set   *set_inscritos,   File_db
*arch_grupos,
557                     File_db *arch_alumnos)
558   {
559         Reg_inscrito baja_inscritos[] = {
560               {1201, 201, 9709},
561               {1201, 201, 9710},
562               {1201, 201, 9711},
563               {0, 0, 0}
564         };
565         Reg_inscrito *p_inscrito;
566         Reg_grupo reg_grupo;
567         Reg_alumno reg_alumno;
568
569         cout << "\n* * * Eliminación de miembros de sets * * *\n";
570         arch_grupos->index("gpo_mat");
571         arch_alumnos->index("matricula");
572         for   (p_inscrito   =   baja_inscritos;   p_inscrito->gpo_cve;
p_inscrito++) {
573               reg_grupo.cve       = p_inscrito->gpo_cve;
574               reg_grupo.mat_cve = p_inscrito->mat_cve;
575               if (!arch_grupos->read_eq(&reg_grupo)) {
576                     cout << " No se localizo grupo:" << reg_grupo <<
"\n";
577                     continue;
578               }
```

```
579                reg_alumno.mat = p_inscrito->alm_mat;
580                if (!arch_alumnos->read_eq(&reg_alumno)) {
581                      cout << "No se localizó el alumno:" << reg_alumno
<< "\n";
582                      continue;
583                }
584                if          (!set_inscritos->remove_member(reg_grupo.ser,
reg_alumno.ser)) {
585                      cout << "Se eliminan del set inscrtos:\n";
586                      cout << "\tOwner: " << reg_grupo << "\n";
587                      cout << "\tmember:" << reg_alumno << "\n";
588                }
589                else {
590                      cout << "Del set set_inscritos se elimino:\n";
591                      cout << "\tOwner: " << reg_grupo << "\n";
592                      cout << "\tmember:" << reg_alumno << "\n";
593                }
594          }
595    }
596
597    //    Prueba de diferentes formas de navegación en el archivo de
alumnos
598    void prueba_navegacion(File_db *arch_alumnos)
599    {
600          Reg_alumno reg_alumno;
601          cout << "\n* * *  Prueba de navegación * * *\n";
602
603          arch_alumnos->index("nombre");
604          cout << "Mayor o igual a Bautista Luis\n";
605          strcpy(reg_alumno.nom, "Bautista Luis");
606          if (arch_alumnos->read_ge(&reg_alumno))
607                cout << "\tEncontrado:" << reg_alumno << "\n";
608          else
609                cout << "\tNo Encontrado:" << reg_alumno << "\n";
610
611          cout << "Mayor que Bautista Luis\n";
612          strcpy(reg_alumno.nom, "Bautista Luis");
613          if (arch_alumnos->read_gt(&reg_alumno))
614                cout << "\tEncontrado:" << reg_alumno << "\n";
615          else
616                cout << "\tNo Encontrado:" << reg_alumno << "\n";
617
618          cout << "Menor o igual que Bautista Luis\n";
619          strcpy(reg_alumno.nom, "Bautista Luis");
620          if (arch_alumnos->read_le(&reg_alumno))
621                cout << "\tEncontrado:" << reg_alumno << "\n";
622          else
623                cout << "\tNo Encontrado:" << reg_alumno << "\n";
624
625          cout << "Menor que Bautista Luis\n";
626          strcpy(reg_alumno.nom, "Bautista Luis");
627          if (arch_alumnos->read_lt(&reg_alumno))
628                cout << "\tEncontrado:" << reg_alumno << "\n";
629          else
630                cout << "\tNo Encontrado:" << reg_alumno << "\n";
631
632          cout << "Menor que Alvarez Martin\n";
633          strcpy(reg_alumno.nom, "Alvarez Martin");
634          if (arch_alumnos->read_lt(&reg_alumno))
```

```
635             cout << "\tEncontrado:" << reg_alumno << "\n";
636         else
637             cout << "\tNo Encontrado:" << reg_alumno << "\n";
638
639         cout << "Mayor que Zamudio Alonso\n";
640         strcpy(reg_alumno.nom, "Zamudio Alonso");
641         if (arch_alumnos->read_gt(&reg_alumno))
642             cout << "\tEncontrado:" << reg_alumno << "\n";
643         else
644             cout << "\tNo Encontrado:" << reg_alumno << "\n";
645 }
```

A continuación se describe el ejemplo. En la descripción las referencias a los números de línea se colocan entre paréntesis.

1). Definiciones de las estructuras registros que se van a usar en el programa (30 - 53).

2). Definición de las funciones operador que imprimen cada uno de los registros (56 - 73).

3). Declaraciones de las funciones contenidas en el programa (75 a 98).

4). El programa principal (101-148) elimina el archivo que contienen la base de datos (104) para crearla a continuación como una base de datos vacía (105).

5). Declaración de los apuntadores a los archivos y sets que constituyen la base de datos (107 - 108)

6). Se invoca a las funciones que crean los archivos y los sets del ejemplo (110 - 115). El parámetro que se pasa a la función es para que la función pueda usar las funciones del objeto Database.

7). Llamados a las funciones que listan y manipulan los archivos (117 -145).

8). La función que crea y carga el archivo de materias (151 - 170) contiene un arreglo con los datos que van a cargar (154 -169). Se define una estructura idx_seg para contener el descriptor del único segmento de la llave (173 -175). Declara un apuntador a un objeto File_db (176) en donde se va a recibir el valor de la función Database::get_file. Se elimina el archivo (179). Se invoca a la función Database::get_file (180) la que nos regresa un apuntador a un objeto File_db. Se crea el índice que corresponde a la clave de la materia (181). Y finalmente en la iteración (182 - 183) se carga la información en el archivo.

9). Las cargas del archivo de alumnos (201 - 222) y de grupos (302 - 323) son similares a la carga de materias.

10). La función lista_materias (189 - 199), establece como llave de acceso para el archivo la clave de la materia (195), asigna a la clave un valor menor (196) y entra a una iteración de lectura y escritura del archivo usando la función File::read_gt la cual localiza el registro con la siguiente llave.

11). Las otras funciones de listado son similares. lista_alumnos_matricula (246 - 256). lista_alumnos_inverso (258 - 269) lista en orden inverso usando la función File::read_lt (266). lista_alumnos_nombre (269 - 279) simplemente utiliza otra llave para realizar la lectura. lista_alumnos_nombre_a_partir_de (280 - 289) se posiciona en un punto arbitrario del archivo de acuerdo al nombre (285). Y finalmente lista_alumnos_orden_fisico (291 - 300) realiza el listado sin utilizar los índices utilizando la función File::start() y File::next.

12). La función crea_inscritos (349 - 403) contiene una tabla de las relaciones propietario - miembro que va a incorporar en el set set_inscritos (353 - 376). Cada elemento de la tabla contiene la llave del archivo de grupos y la llave del archivo de alumnos. Primeramente se elimina el set set_inscritos (381). A continuación se invoca a la función Database::get_set (383) para crear el set. Posteriormente se establecen los índices que se van a usar para leer estos dos archivos (386 - 387) y finalmente, en la iteración (388 - 401) se leen los registro de ambos archivos y se invoca la función Set::add_member_end (400) que incorpora un nuevo miembro a un set.

13). La función lista_de_grupo (407 - 424) contiene un ciclo while para leer secuencialmente el archivo de grupos (416). Dentro de la iteración se lee el primer miembro del set definido por el registro propietario actual (418). Si se localiza, se entra a un ciclo do - while (419 - 421) para continuar leyendo el resto de los registros miembros del set.

14). La función tiras_de_materia (426 - 450) es similar a la anterior pero en este caso primeramente se lee el registro del archivo miembro del set (438) posteriormente se lee el primer registro propietario (440) y si se localiza se continua leyendo el resto de los registros propietarios.

15). La función baja-de_alumnos (453 - 474) elimina a los alumnos contenidos en una tabla. Estos alumnos ya forman parte del set set_inscritos por lo tanto el sistema tendrá que eliminar todas sus referencias. La iteración for (467 - 473) recorre la tabla de los alumnos que se van a eliminar. El registro a eliminar se lee (469) y si la lectura tiene éxito, el registro se da de baja (472).

16). La función creación_requisitos (476 - 518) es similar a la creación del set set_inscritos. La diferencia estriba en que en este set el archivo propietario y el

archivo miembro son el mismo. Las funciones lista_requisitos (521 - 536) y lista_requisitos_de (538 - 553) leen los registros miembros de un set y los registro propietarios del mismo registro miembro.

17). La función elimina_alumnos_de_grupo (555 - 595) elimina miembros del set set_inscritos. La tabla baja_inscritos (559 - 564) contiene las llaves de los registros propietarios y registros miembros del set. Dentro de la iteración se leen ambos registros (575 - 583) y con los identificadores seriales de ambos registros se invoca la función Set::remove_member (584) para eliminar un miembro del set.

18). La función prueba_navegacion (597 - 645) sirve para probar las funciones de lectura relativas al registro actual.

8.2 Demo2

Este programa tienen como propósito probar la permanencia de la información en la base de datos. Utiliza la base de datos generada por el programa de prueba Demo1 y realiza las siguientes operaciones.

1). Abre la base de datos ESCOLAR.DB.

2). Abre los archivos: MATERIAS.DAT, ALUMNOS.DAT, GRUPOS.DAT.

3). Abre los sets: INSCRITOS.DAT y REQUIS.DAT.

4). Efectúa las mismas explotaciones de programa DEMO1, las cuales son:

- Listado de los archivos de materias.
- Listado de alumnos por matrícula.
- Listado de alumnos por nombre.
- Listado de alumnos por nombre a partir de un punto inicial.
- Listado de alumnos por orden fisico
- Listado de grupos
- Listas de grupo
- Tiras de materia
- Listado de los requisitos de una materia
- Listado de las materias para las cuales una materia es requisito.

Las funciones de este programa son exactamente las mismas que las del programa anterior. Solamente se han eliminado las funciones de creación de los archivos y sets.

8.3 demo3

El programa demo3 realiza una prueba de volumen. En este programa se generaron 50,000 registros con tres llaves una de las cuales es concatenada. El programa tiene capacidad de generar hasta 250,000 registros. Con esta prueba se asegura la integridad de las estructuras de índices con varios niveles. Los índices generados resultaron de tres niveles. A continuación se presenta el listado del programa:

```
// ejemplo7.h
#include <iostream.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "database.h"
typedef     struct {
    public:
      long  ser;
      long  mat;
      char  nom[ 50] ;
      char  dom[ 51] ;
} alm_r;
ostream &operator << (ostream &s, alm_r &r)
{

      return s << r.ser << "." << r.mat << "."   << r.nom
            << "     " << r.dom << "\n";
}

class Gen_alm {
    private:
      alm_r alm;
      int   num_ap;
      int   num_nom;
      long  serial;
      int   i, j, k;
    public:
      Gen_alm();
      void  otro(alm_r *al);
};
```

```
// Ejemplo7.cpp
#include "demo3.h"
char *apellidos[ ] = {
      "Manrique", "Marín",    "Marquez",    "Martel",    "martín",
      "Martinez", "Mayoral", "Medina",     "Mendez",    "Mendoza",
      "Miranda",  "Moncada", "Moncayo",    "Morales",   "Muñoz",
      "Najera",   "Nava",     "Navarro",    "Neri",      "Nieto",
      "Nolasco",  "Noriega", "Novelo",     "Nuñez",     "Ocaña",
      "Olguín",   "Oliva",     "Olivares", "Olmedo",    "Olvera",
      "Orozco",   "Ortega",    "Ortiz",    "Padilla",   "Padua",
      "Pagaza",   "Palacios", "Paniagua", "Pantoja",   "Paramo",
      "Paredes",  "Pastrana", "Paulin",    "Paz",       "Pelayo",
      "Peniche",  "Perales",  "Perez",     "Piña",      "Plascencia",
      "Ponce",    "Porraz",   "Portilla", "Pozos",     "Prieto",
      "Quintanilla","Quiroz", "Rábago",    "Ramirez",   "Reyes",
      "Robles",   "Romero",   "Ruiz",      "Salazar",   "Sanabria",
      "Sánchez",  "Sierra",   "Soto",      "Suarez",    "Tavera"
```

```
18  };
19  char *nombres[] = {
20        "José Luis","Marcos",    "María",  "Magdalena",  "Rosario",
21        "Dagoberto","Linda",     "Alfredo",  "José María","Aida",
22        "Antonio",  "Araceli",   "Guadalupe","Berta",     "Ismael",
23        "Ulises",   "Ezequiel",  "Mario",    "Abel",      "Juan",
24        "Anacleto", "Anselmo",   "Ariadna",  "Alondra",   "Ana María",
25        "Andrea",   "Cristina",  "Elena",    "Catalina",  "Consuelo",
26        "Braulio",  "Benito",    "Blanca",   "Alicia",    "Carmen",
27        "Bonifacio","Concepción",     "Cruz","Calixto",  "Carolina",
28        "Dario",    "Diógenes",  "Dionisio", "Dante",     "Ernesto",
29        "Enrique",  "Esperanza","Eleuterio","Flor",      "Federico",
30        "Felipe",   "Fidias"
31  };
32
33  char *domicilios[] = {
34        "Independencia # 16",
35        "Juarez 11",
36        "San Pablo 180",
37        "Venustiano Carranza 121"
38        "22 de Febrero 211",
39        "Presa de la Angostura 114",
40        "Pafnuncio Padilla 25",
41        "Lazaro Cárdenas 120",
42        "Clavería 34",
43        "5 de Mayo 20",
44        "Azcapotzalco 120",
45        "Centenario 405",
46        "Allende 45",
47        "Victor Hugo 33",
48        "Regina 34"
49  };
50
51  Gen_alm::Gen_alm()
52   {
53
54        num_ap = sizeof(apellidos) / sizeof(apellidos[ 0] );
55        num_nom = sizeof(nombres) / sizeof(nombres[ 0] );
56        serial = 1;
57        i = j = k = 0;
58  }
59  void Gen_alm::otro(alm_r *al)
60  {
61        if (k >= num_nom) {
62              k = 0;
63              j++;
64              if (j >= num_ap) {
65                    j = 0;
66                    i++;
67                    if (i >= num_ap) {
68                          cout << "Error. demasiaodos nombres\n";
69                          exit(1);
70                    }
71              }
72        }
73
74        strcpy(al->nom, apellidos[ i] );
75        strcat(al->nom, " ");
76        strcat(al->nom, apellidos[ j] );
```

```
  77        strcat(al->nom, ",");
  78        strcat(al->nom, nombres[ k] );
  79        al->mat = serial;
  80        strcpy(al->dom,domicilios[ serial%(sizeof(domicilios)/
sizeof(domicilios[ 0] ))] );
  81        serial++;
  82        k++;
  83  }
  84  void main()
  85  {
  86        remove("Alumnos.dat");
  87        File alumnos("Alumnos.dat");
  88
  89        alm_r al;
  90        Gen_alm  gg;
  91        long i;
  92
  93        struct idx_seg alm_dom_ser[ ] = {
  94              { (char*)al.dom-(char*)&al, sizeof(al.dom),
SEG_TYPE_CHAR, 0} ,
  95              { (char*)&al.ser-(char*)&al, sizeof(al.ser),
SEG_TYPE_SERIAL, 0} ,
  96        } ;
  97
  98        struct idx_seg alm_is[ ] = {
  99              { (char*)&al.mat-(char*)&al, sizeof(al.mat),
SEG_TYPE_LONG, 0}
 100        } ;
 101
 102        struct idx_seg alm_nom[ ] = {
 103              { (char*)&al.nom-(char*)&al, sizeof(al.nom),
SEG_TYPE_CHAR, 0}
 104        } ;
 105        alumnos.createx("domicilio", 2, alm_dom_ser);
 106        alumnos.createx("matricula", 1, alm_is);
 107        alumnos.createx("nombre", 1, alm_nom);
 108        for(i = 0; i < 50000; i++) {
 109              gg.otro(&al);
 110              if (!alumnos.write(&al, sizeof(al)))
 111                    fatal_error("Llave duplicada");
 112              if (i % 100 == 0)
 113                    cout << "Grabados:" << i << "\n";
 114        }
 115  }
```

1). En el archivo demo3.h se define un tipo de datos alm_r (7 - 13) que contiene el formato del registro del archivo que se va a generar.

2). Contiene la definición de la función operador << para imprimir los datos del registro.

3). La clase Gen_alm (21 - 31) sirve para generar los registros de prueba.

4). El archivo demo3.cpp contiene una tabla de apellidos (3 - 18), una de nombres (19 - 31) y una tabla de domicilios (33 - 49) con las cuales se van a generar los nombres que se van a incluir en los registros.

5). La función constructora de Gen_alm (51 - 58) inicializa las variables del objeto.

6). La función Gen_alm::otro (59 - 72) genera el siguiente registro que se va a grabar.

7). El programa principal (84 - 115) contiene las estructuras que describen las llaves de los índices (93 - 104). La primera llave es el domicilio (93 - 96). Como el domicilio se duplica la llave tiene un segmento serial adicional para hacer la llave única. La segunda llave es la matrícula (98 - 100) y la tercera es el nombre del alumno (102 - 104). El programa es muy simple, se crean los índices (105 - 107) . En la iteración for (108 - 114) se generan y se graban los registros.

8.4 demo4

Este programa toma el archivo generado en demo3 y realiza lecturas secuenciales por cada una de las llaves. Se listan los primeros quinientos registros por cada llave.

```
1  #include "demo3.h"
   2  #include <stdio.h>
   3  void main()
   4  {
   5        long i;
   6        File alumnos("Alumnos.dat");
   7        alm_r al;
   8
   9        alumnos.index("matricula");
  10
  11        al.mat = 0;
  12        if (alumnos.read_ge(&al))
  13              for (i = 0; i < 500; i++) {
  14                    cout << al << "\n";
  15                    if (!alumnos.read_gt(&al))
  16                          break;
  17              }
  18
  19        al.mat = 50000;
  20        if (alumnos.read_le(&al))
  21              for (i = 0; i < 500; i++) {
  22                    cout << al << "\n";
  23                    if (!alumnos.read_lt(&al))
  24                          break;
  25              }
  26
  27        strcpy(al.dom, " ");
  28        alumnos.index("domicilio");
  29        if (alumnos.read_ge(&al))
  30              for (i = 0; i < 500; i++) {
  31                    cout << al << "\n";
  32                    if (!alumnos.read_gt(&al))
  33                          break;
  34              }
  35  }
```

8.5 demo5

El programa demo5 genera un archivo con una llave concatenada con seis segmentos. La llave está de tal manera construida que provoca que los segmentos numéricos queden desalineados.

```
1  // Demo5.- Prueba de llaves concatenadas
   2
   3  #include "database.h"
   4  #include <iostream.h>
   5  #include <stdio.h>
   6  struct s_fict {
   7        char c1;
   8        double c2;
   9        char c3;
```

```
10          long c4;
11          char c5;
12          int c6;
13    };
14    ostream& operator<<(ostream& s, struct s_fict& a)
15    {
16          s << a.c1 << "-" << a.c2 << "-" << a.c3 << "-"
17            << a.c4 << "-" << a.c5 << "-" << a.c6 << endl;
18          return s;
19    }
20    void genera(struct s_fict& s)
21    {
22          static n = 0;
23          s.c1 = s.c3 = s.c5 = 'a' + n % 26;
24          s.c2 = (double)n;
25          s.c4 = (long)n;
26          s.c6 = n;
27          n++;
28    }
29    void main()
30    {
31          struct s_fict s;
32          int i;
33
34          remove("concat.dat");
35          File f("concat.dat");
36          struct idx_seg llave[] = {
37                { (char*)&s.c1-(char*)&s,   sizeof(s.c1),   SEG_TYPE_CHAR,
0} ,
38                { (char*)&s.c2-(char*)&s,  sizeof(s.c2),  SEG_TYPE_DOUBLE,
0} ,
39                { (char*)&s.c3-(char*)&s,   sizeof(s.c3),   SEG_TYPE_CHAR,
0} ,
40                { (char*)&s.c4-(char*)&s,   sizeof(s.c4),   SEG_TYPE_LONG,
0} ,
41                { (char*)&s.c5-(char*)&s,   sizeof(s.c5),   SEG_TYPE_CHAR,
0} ,
42                { (char*)&s.c6-(char*)&s, sizeof(s.c6), SEG_TYPE_INT, 0}
43          };
44
45
46
47          f.createx("llave",6, llave);
48          for (i=0; i < 1000; i++) {
49                genera(s);
50                f.write(&s, sizeof(s));
51          }
52
53          s.c1 = ' ';
54          while(f.read_gt(&s))
55                cout << s;
56
57          s.c1 = 'z'+1;
58          while(f.read_lt(&s))
59                cout << s;
60    }
```

9. Conclusiones

Con el desarrollo de UAMRM se lograron todos los objetivos inicialmente planteados. Se cuenta con un manejador de archivos que satisface los requerimientos iniciales de almacenamiento de información planteados en el proyecto de robótica. con las siguientes características:

1. Facilidad de uso. Los llamados a las funciones para el manejo de la información son sumamente simples.

2. Se cuenta con los fuentes totalmente documentados para poder realizar futuras adaptaciones a los nuevos requerimientos que se planteen.

3. Durante el desarrollo el sistema se probó en ambiente MS-DOS, WINDOWS 95, LINUX y UNIX Solaris. No vemos dificultad para adaptarlo a otras plataformas.

4. En cuanto a la eficiencia podemos mencionar los siguientes datos:

   a). La bilioteca de objetos ocupa un total de 146 KB en ambiente Windows 95.

   b). El tiempo de ejecución para la carga de 50,000 registros de 158 bytes con tres llaves, una de 4 bytes y otra de 50 bytes tomó 4 minutos 24 segundos. La prueba se llevó a cabo en una PC con procesador 486 a 66 Mhz en ambiente Windows 95.

   c). El espacio utilizado para el almacenamiento de los 50,000 registros antes mencionados con sus llaves fue de 14.4 MB, por lo tanto, el espacio requerido por registro fue de 288 bytes. Si consideramos como información la longitud del registro y las llave tenemos 212 bytes por registro con lo cual se obtiene una eficiencia de 73.6 %.

10. Perspectivas futuras

Un manejador de archivos es la infraestructura básica para la construcción de un sistema manejador de bases de datos. En el caso particular del manejador UAMRM mencionamos a continuación las posibilidades futuras de desarrollo:

1). Agregar cierto nivel de seguridad con el manejo de log de transacciones para recuperación de la información en caso de falla.

2). Agregarle lógica para proporcionar acceso simultáneo a varios usuarios.

3). Agregar un diccionario de datos para almacenar la información relativa a las estructuras de los registros y archivos de una base de datos.

4). Convertirlo en una aplicación cliente - servidor a fin de que el manejador pueda residir en un equipo de la red y los usuarios lo puedan invocar desde otros equipos.

5). Diseñar un lenguaje de alto nivel para los usuarios. Un lenguaje similar a SQL con algunas ampliaciones para el manejo de sets.

11. Bibliografía

[1] KHACHATUROV, GEORGII; GONZÁLEZ, SILVIA; IBARRA, JUAN MANUEL; MONCAYO, HUGO. Diseño conceptual de un sistema de aprendizaje visual. Memorias del Simposium Internacional de Computación, Centro Nacional de Cálculo del Instituto Politécnico Nacional. Noviembre de 1995.

[1] KHACHATUROV,GEORGII; GONZÁLEZ, SILVIA. A Technology Based Decomposition of Robot Task into Elementary Problems. Reporte de investigación en preparación.

[2] CODASYL *Data Base Task Group Report*, 1971. Association for Computing Machinery, Nueva York, abril 1971.

[3] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W. *Object-Oriented Modelling and Design.* Prentice Hall, Engelwood Cliffs, New Jersey, 1991

[4] Schildt, Herbert. C++ *Guía de Autoenseñanza.* MaGraw-Hill Interamericana de España, S. A. 1995

[5] YOURDON, EDWARD. *Análisis Estructurado Moderno.* Prentice-Hall Hispanoamericana (1993).

[6] Codd, E.F., A Relational Model of Data for Large Shared Data Banks. Communications of the ACM, vol 13, No. 6, junio 1970.

[7] COMER, DUGLAS. "The Ubiquitous B-Tree". *Computing Surveys* Vol 11, No. 2 (Junio 1979) 121-137.

[8] DATE, C. J.An Introduction to Database Systems. Sixth Edition. Addison-Wesley Publishing Company, Inc 1995.

APÉNCICE. Glosario de términos

ancla. Un registro en un set que sirve para localizar el principio de una cadena.

archivo. Un conjunto de registros de datos y sus estructuras de índices.

archivo_bd. Un conjunto de registros de datos y estructuras de índices que puede formar parte de un **set** en una base de datos

archivo de datos. Un archivo que contiene información del usuario y que el usuario puede manipular directamente.

base de datos. Estructura contiene los conjuntos y archivos_bd

bloque. La unidad de información que se lee o escribe físicamente en un archivo.

bloque de datos. Contiene uno o varios registros de datos.

bloque de índices. Uno de los bloques de un archivo que pertenece a la estructura de árbol B. Contiene llaves, apuntadores a los bloques de datos y apuntadores a bloques de índices del siguiente nivel.

bloque descriptor. Primer bloque de un archivo (bloque 0) que contiene la descripción del archivo y de sus estructuras de índices.

bloque libre. Es un bloque que como resultado de las eliminaciones de registros de datos o llaves queda libre dentro del archivo.

buffer. Unidad de memoria que contiene un bloque además de un conjunto de datos para administrarlo.

bytes de alineación. Son los bytes que se dejan sin uso a fin de que el siguiente campo de la llave se alinee a una dirección de memoria apropiada para el tipo de dato.

caché de buffers. Un mecanismo de software que administra los buffers del sistema y que optimiza el tiempo de acceso a los mismos.

cadena miembro. Una lista ligada de todos los eslabones que hacen referencia a los registros propietarios de un mismo registro miembro.

cadena propietaria. Una lista ligada de todos los eslabones que hacen referencia a los registros miembros del mismo propietario en un set.

eslabón. Uno de los registros que representan una relación propietario - miembro en un set. Estos registros forman las cadenas propietarias y las cadenas miembro

índice. Una estructura de árbol B que se utiliza para localizar un registro de datos mediante una llave de acceso.

índice actual. El índice que ha sido seleccionado para que las operaciones de entrada y salida que utilizan un índice lo hagan con él.

instancia de un set. Una relación entre un registro del archivo propietario del set y uno o varios registros del archivo miembro del set.

llave. Un dato o conjunto de datos con los que se puede identificar de manera única un registro dentro de un archivo.

llave segmentada. Una llave compuesta por varios datos elementales.

segmento. Uno de los datos elementales que constituye una llave.

Serial. Tipo de dato que se usa para formar llaves que genera un número secuencial único para cada registro cuando se graba en el archivo.

registro. Un conjunto de datos que participan en una operación de entrada o salida en el sistema.

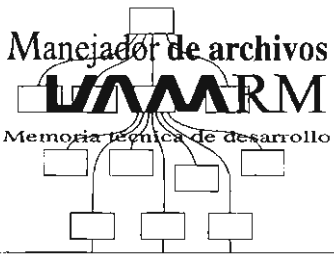
registro actual. El registro grabado, leído o modificado más recientemente en un archivo.

registro miembro. Un registro que aparece como subordinado en la jerarquía de un set.

registro propietario. Un registro que aparece como propietario en la jerarquía de un set.

set. Conjunto de relaciones entre los registros de un archivo propietario y un archivo miembro. Cada relación en una instancia del set.

Índice de Funciones



Se terminó de imprimir en el mes de julio del año 2001 en los talleres de la Sección de Impresión y Reproducción de la Universidad Autónoma Metropolitana, *Unidad Azcapotzalco*

La edición estuvo a cargo de la Sección de Producción y Distribución Editoriales

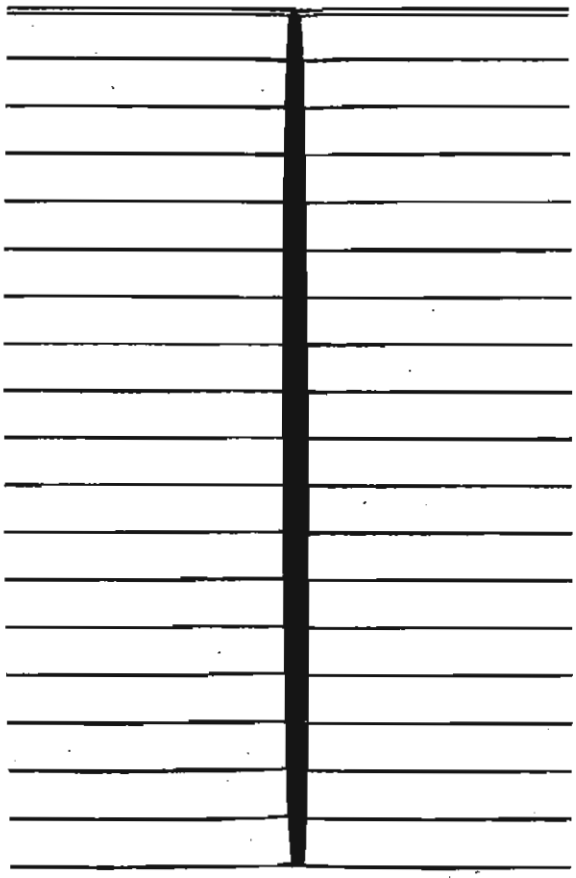
Se imprimieron 100 ejemplares más sobrantes para reposición.

Formato de Papeleta de Vencimiento

El usuario se obliga a devolver este libro en la fecha señalada en el sello mas reciente

Código de barras. 2893176

FECHA DE DEVOLUCION



- Ordenar las fechas de vencimiento de manera vertical.
- Cancelar con el sello de "DEVUELTO" la fecha de vencimiento a la entrega del libro

UNIVERSIDAD AUTÓNOMA METROPOLITANA
Casa abierta al tiempo Iztapalapa

División de Ciencias Básicas e Ingeniería
Departamento de Sistemas

Coordinación de Extensión Universitaria
Sección de Producción y Distribución Editoriales