DOCUMENTACIÓN DEL SISTEMA GESTOR DE BASES DE DATOS

Con el propósito de agilizar la búsqueda y obtención de información acerca del código del sistema de gestor de bases de datos se ha dividido a la presente documentación en dos secciones:

1. Sección integradora: explica el funcionamiento global del sistema, así como también aporta una ruta recomendada para la lectura de la sección acerca de las funciones.
2. Sección de documentación de las funciones: Lista todas las funciones en el código de manera **alfabética**, teniendo un formato similar al de un diccionario. El propósito de presentar la información de esta manera es hacer más fácil la búsqueda de ciertas funciones, aunque en el formato electrónico siempre se puede optar por hacer una búsqueda en el documento con la porción de texto a buscar.

Aclarado lo anterior continuamos con la explicación del funcionamiento del sistema gestor de bases de datos

El sistema gestor de bases de datos se compone de varios archivos en lenguaje C, los cuales se listan a continuación:

* SGBD.c → Se trata del script donde se encuentra la función principal void main() del sistema gestor de bases de datos. También cuenta con una función debug() cuyo propósito es el de probar nuevas implementaciones que estén en desarrollo, resolver bugs (debugging), además de proporcionar una disposición visual separada de aquel código que no sea con propósito de debugging.
* tablas.h → Es un archivo cuyo código es incorporado como una librería en SGBD.c, en el cual podemos encontrar a aquellas funciones relacionadas con la gestión de tablas, lo cual implica su creación, modificación y eliminación, además de proporcionar asistentes guiados de manipulación de tablas al usuario
* disco.h → Archivo incorporado como librería en SGBD.c cuyo propósito es proporcionar todas las funciones relacionadas con escribir o leer en disco, lo cual abarca recuperar información previamente almacenada en disco o disponer de nueva información para guardarla en el disco
* crud.h → Archivo incorporado como librería en SGBD.c cuyo nombre es acrónimo de Create, Read, Update, Delete (CRUD), y que contiene todas las funciones relacionadas con agregar, modificar, leer o eliminar nuevos registros en las tablas. Su diferencia con tablas.h es que tablas.h se dedica a tareas de gestión de tablas mientras que crud.h hace gestión de registros.

La razón de dividir el código en varios archivos, así como utilizar un enfoque de funciones que sean muy modulares es seguir el principio KISS (Keep It Simple, Stupid!), una filosofía de diseño que aboga por evitar la complejidad a toda costa. Tener funciones muy modulares permite hacer un debugging más sencillo por lo que ha de seguirse este principio en aras de simplificar el trabajo.

Cualquier modificación y nueva implementación debe ser reportada y correctamente explicada en la documentación de SGBD, tanto en la sección integradora (explicando su relación con el funcionamiento global del sistema) y en la sección de funciones (en donde se detalla el funcionamiento de la nueva implementación).

**SECCIÓN INTEGRADORA**

Entonces, tenemos que la función SGBD.c luce de esta manera normalmente

#include<stdio.h>

#include<stdlib.h>

#include "tablas.h"

#include "disco.h"

#include "crud.h"

// Sistema gestor de bases de datos

void debug(){

}

void main(){

debug();

}

La función main posee una función debug() destinada a probar implementaciones y resolver errores

* *Ver* ***debug()***

Es importante aclarar que el orden en que están incorporadas las directivas del tipo #include es importante, pues recordemos que el preprocesador al encontrar #include “librería.h” sustituye dicha línea por la totalidad del código del archivo “librería.h”, además de que una librería B necesitan recursos y variables que están definidas en otra librería A, de manera que es necesario usar la directiva #include primero con la librería A y luego con la librería B

Por otro lado, el SGBD se compone de tablas Una tabla es un objeto conceptual que permite el almacenamiento de información y a pesar de la diversidad en el tipo de datos que pueden guardar, estas cuentan siempre con características comunes entre ellas (al menos para el propósito de nuestro sistema gestor de bases de datos).

Lo anterior, denota la necesidad de implementar una estructura “tabla” con atributos que sabemos que toda tabla tiene, sin importar el valor numérico, escalar o nominal de dichos atributos.

Queda definida la estructura tabla en tablas.h del siguiente modo:

struct tabla{

char \*nombre; // Consiste en el nombre de la tabla

int nCampos; // La cantidad de campos en la tabla

char \*\*metadatos; // Almacena los nombres de todos los campos

void \*\*\*registros; // Almacena los datos guardados en la tabla

};

typedef struct tabla Tabla; // Definimos el sinónimo Tabla

Las tablas necesitan ser creadas, por eso se implementa una función crearTabla, la cual, como todas las funciones de aquí en adelante, pueden ser consultada en la sección de funciones

* *Ver* ***crearTabla***

Sin embargo, en nuestro programa no aspiramos a almacenar una sola tabla, sino varias de ellas, las que necesite el usuario, por lo que enfrentamos la necesidad de una implementación de apuntador a estructura Tabla, un tipo de variable que nos hará posible almacenar varias tablas en un arreglo de estructuras Tabla.

En el presente SGBD hemos nombrado al conjunto de Tablas como “espacio de Tablas”, siendo este un espacio en memoria para almacenar todas las tablas. En el código hay una función dedicada a crear el espacio de Tablas

* *Ver* ***generarEspacioTablas***

Asimismo hay ocasiones en donde enfrentaremos la necesidad de ampliar o disminuir el espacio para tablas, conforme se eliminen o se agreguen nuevas tablas, para lo cual usaremos la función ***redimEspacioTablas*** (redimensionar espacio para tablas), dentro de la cual se hace uso de la función nativa ***realloc*** que permite cambiar el tamaño de un espacio de memoria asignado dinámicamente a un apuntador (para usar realloc necesitamos la ldirectiva #include<stdlib.h>)

* *Ver* ***redimEspacioTablas***

Con el propósito de redimensionar las tablas, necesitamos saber el tamaño actual de la tabla, de manera que si se agregan tablas o se eliminan podamos cambiar el tamaño de dichas tablas. Para ello usamos la función enunciada a continuación:

* *Ver* ***getSizeEspacioTablas***

Con las anteriores funciones ya fue posible crear una función que hiciera más cosas de manera que se cumpla el principio KISS, de modo que se implementa un asistente de creación de Tablas, el cual puede ser consultado en su respectivo apartado en la sección de funciones. Usamos esta función para proporcionar un entorno guiado al usuario para crear nuevas tablas. Nótese la implementación de la función *strdyread()*

* *Ver* ***asistCreacionTablas***

Asimismo, se ha incorporado una función que se dedica exclusivamente a leer los nombres de cada uno de los metadatos para la nueva Tabla a crear

* *Ver* ***nombrarCampos***

A partir de esta función, tendremos que para leer cadenas, en vez de usar alguna función nativa de las librerías del compilador de C tales como *scanf, gets, fgets,* entre otras, haremos uso de una función personalizada llamada *strdynread()* [String Dynamic Read] la cual se vale de la función realloc para leer cadenas de forma completamente dinámica, sin recurrir a arreglos auxiliares de tamaño fijo

* Ver ***strdynread()***

La manera tradicional de leer cadenas con scanf es

*scanf(“%s”, cadena);*

Sin embargo la manera de hacerlo con *strdynread()* es

*cadena=strdynread();*

Otro propósito del SGBD es ser capaz de guardar todos los datos almacenados en el disco de la computadora, de manera que puedan ser accesibles posteriormente. Las utilidades que permiten dicho propósito están en el archivo disco.h

De manera análoga a como empezamos con funciones que operaban con tablas y gradualmente fuimos a los espacios de tablas en un dominio superior, de la misma manera se abordará la documentación de las funciones para almacenamiento en disco

Se hace necesaria la implementación de una función que permite exclusivamente guardar una tabla a la vez en disco y bajo un cierto formato que haga fácil su posterior lectura. Toda la información será almacenada en un archivo *tablas.dat*. Sin embargo, esta función solo guarda el encabezado de la tabla, es decir, su nombre, cantidad de campos, y el nombre de cada metadato pues se ocupa otra función para guardar los registros (función pendiente)

El formato con el cual se guardan las tablas en *tablas.dat* luce de este modo:

*==== TABLE ====*

*tabla1*

*3*

*ID*

*nombre*

*apellido*

*&*

*==== TABLE ====*

*tabla2*

*4*

*ID Producto*

*Nombre producto*

*Precio*

*Costo unitario*

*&*

Analizando

*==== TABLE ==== → Es un separador entre tablas*

*tabla1 → nombre de la tabla*

*3 → numero de campos/metadatos*

*ID → nombre del metadato #1*

*nombre → nombre del metadato #2*

*apellido → nombre del metadato #3*

*& → carácter delimitador final de tabla*

*==== TABLE ==== → Segunda tabla*

*tabla2*

*4*

*ID Producto*

*Nombre producto*

*Precio*

*Costo unitario*

*&*

* *Ver* ***guardarTabla***

Disponemos de una función que permita guardar el espacio de tablas completo, tal que haga una llamada a *guadarTabla* para cada tabla en el espacio de tablas

* *Ver* **guardarEspacioTablas**

Necesitamos también una función que sea capaz de leer en disco una tabla [Función en desarrollo]

* *Ver* **leerTabla**

En leerTabla ocupamos que lea la totalidad de una línea hasta encontrarse con un salto de línea o con el final del archivo. Esta función es similar a strdynread() pero trabaja sobre archivos. Se enuncia a continuación

* *Ver* ***dynFileread***

**SECCIÓN DE DOCUMENTACIÓN DE FUNCIONES**

**asistCreacionTablas → tablas.h**

**La función mostrada a continuación no recibe ningún parámetro, pues se dedica a generar los datos que necesitan las funciones en su interior. Sirve como asistente del usuario para la creación de Tablas**

// Función asistente de creación de tablas en la interfaz del usuario

Tabla \*asistCreacionTablas(){

int u,nCampos;

char \*nombre,\*\*metadatos;

// Le solicitamos al usuario el nombre de la tabla

printf("Nombre de la tabla: \n");

// Leemos ese nombre mediante strdynread()

nombre=strdynread();

// Limpiamos el buffer

fflush(stdin);

// Le pedimos al usuario el número de campos

printf("Elija el número de campos en la nueva tabla: \n");

scanf("%d",&nCampos);

// getchar permite consumir un salto de línea que queda atorado en el buffer en la anterior ejecución de scanf, es un getchar quimera para el buen funcionamiento del programa

getchar();

// Usamos la función ***nombrarCampos*** *pa*ra pedirle al usuario los nombres de los campos y los guardamos en un apuntador doble a cadena “metadatos” (un arreglo de cadenas que contiene los nombres de cada metadato para la nueva Tabla

metadatos=nombrarCampos(nCampos);

// Una vez que ya recolectamos todos los datos necesarios, los enviamos a crearTabla y posicionamos toda la información de la nueva tabla en un apuntador a Tabla newTabla

Tabla \*newTabla=crearTabla(nombre,nCampos,metadatos);

// Devolvemos la nueva tabla recién creada

return newTabla();

}

**crearTabla → tablas.h**

**Esta función recibe un (char \*) nombre, una cantidad (int) de Campos y un conjunto (char \*\*) de varios apuntadores a apuntadores a carácter, es decir, un arreglo de cadenas.**

Tabla \*crearTabla(char \*nombre,int nCampos,char \*\*metadatos){

// Asignamos memoria para un apuntador a estructura Tabla

Tabla \*tabla=(Tabla \*)malloc(sizeof(Tabla));

// Asignamos memoria para cada atributo de la estructura y le asignamos su valor copiando una cadena a otra mediante strcpy()

tabla->nombre=(char \*)malloc(sizeof(char)\*strlen(nombre));

strcpy(tabla→nombre,nombre);

tabla->nCampos=nCampos;

tabla->metadatos=(char \*\*)malloc(sizeof(char \*)\*nCampos);

// Asignamos memoria para cada metadato de tabla y le asignamos su respectivo nombre

for(int u=0;u<nCampos;u++){

tabla->metadatos[u]=(char \*)malloc(sizeof(char)\*strlen(metadatos[u]));

strcpy(tabla->metadatos[u],metadatos[u]);

}

// Devolvemos el apuntador a tabla de la Nueva tabla creada

return tabla;

}

**debug() → SGBD.c**

**Esta función funciona como un recipiente. Su uso es para escribir código que permita probar el correcto desempeño de las características implementadas, por lo que no tiene un código específico. Usar una llamada a debug() en el main() de SGBD.c**

**dynFileread() → disco.h**

**Recibe un apuntador a archivo y lee una línea completa del archivo hasta toparse con un salto de línea o bien con el final del archivo (EOF, End Of File)**

char \*dynFileread(FILE \*f){

// Declaramos la cadena que vamos a devolver en return al final

char \*cad=NULL;

// Declaramos un índice ind=0 y un entero c con valor igual a EOF

int ind=0,c=EOF;

// Mientras c no valga 0

while(c){

// Obtener el carácter en turno

c=fgetc(f);

// Si el carácter es un salto de línea o EOF entonces haz que c=0

// Entonces el siguiente bucle no se ejecutará al ser c=0

if(c=='\n' || c==EOF) c=0;

// Aumenta el tamaño del bloque de memoria del apuntador cad en 1

cad=realloc(cad,ind+1);

// Asigna el nuevo carácter c a cad en la posición actual “ind” y después incrementa en 1 esa posición actual “ind”

cad[ind++]=c;

}

// Regresa la nueva cadena

return cad;

}

**generarEspacioTablas → tablas.h**

**La función recibe un entero m que representa la cantidad de Tablas que podrá guardarse en memoria del espacio para Tablas.**

Tabla \*\*generarEspacioTablas(int m){

Tabla \*\*conj=(Tabla \*\*)malloc(sizeof(Tabla \*)\*m);

return conj;

}

**getSizeEspacioTablas → tablas.h**

**La función solamente recibe el apuntador doble a Tabla (el espacio de tablas) conj del cual vamos a examinar la cantidad de Tablas que contiene**

int getSizeEspacioTablas(Tabla \*\*conj){

int u;

for(u=0;conj[u];u++);

return u;

}

**guardarEspacioTablas**

**La función recibe un espacio de tablas y va guardando cada tabla en el espacio de tablas al hacer una llamada a guardarTabla para cada una de ellas**

void guardarEspacioTablas(Tabla \*\*espacio){

int u;

for(u=0;espacio[u];u++)

guardarTabla(espacio[u]);

}

**guardarTabla → disco.h**

**Mediante esta función guardamos cada uno de los datos de una estructura Tabla, a excepción de los registros. Se puede decir que guarda el encabezado de la Tabla.**

void guardarTabla(Tabla \*tabla){

// Se inicializa un apuntador a archivo

FILE \*f=fopen("tablas.dat","at");

// Escribimos en el archivo un separador e identificador de una nueva Tabla

fprintf(f,"==== TABLE ====\n");

// Escribimos posteriormente el archivo el nombre y cantidad de campos

fprintf(f,"%s\n",tabla->nombre);

fprintf(f,"%d\n",tabla→nCampos);

// Escribimos el nombre de cada campo/metadato

for(int u=0;u<tabla->nCampos;u++)

fprintf(f,"%s\n",tabla→metadatos[u]);

// Escribimos el carácter delimitador de cada tabla

fputc('&',f);

// Escribimos un salto de línea

fputs('\n',f);

// Cerramos la conexión con el apuntador a archivo ‘f’

fclose(f);

}

**leerTabla → disco.h [FUNCIÓN PENDIENTE DE PROBAR Y CORREGIR ERRORES)**

**Su función es capturar todos los datos de una tabla (a excepción de sus registros) para almacenarlos en la memoria RAM, es decir, que el programa en ejecución posea estos datos**

Tabla \*leerTabla(){

int ind,c;

char \*\*metadatos,\*nombre;

int nCampos;

FILE \*f=fopen("tablas.dat","rt");

nombre=dynFileread(f);

fscanf(f,"%d\n",&nCampos);

while(c!='&'){

metadatos=realloc(metadatos,sizeof(char \*\*)\*(ind+1));

metadatos[ind]=dynFileread(f);

ind++;

}

Tabla \*tabla=crearTabla(nombre,nCampos,metadatos);

return tabla;

}

**main() → SGBD.c [EN DESARROLLO]**

**Flujo de ejecución principal del programa. Es la función más compleja del programa, al poseer acceso directo o indirecto a todas las funciones implementadas.**

**nombrarCampos → tablas.h**

**Recibe la cantidad de campos (lo que es lo mismo, metadatos), para saber la cantidad de nombres que vamos a pedir para cada campo/metadato**

char \*\*nombrarCampos(int nCampos){

int u=0;

char \*\*metadatos=(char \*\*)malloc(sizeof(char \*)\*nCampos);

for(u=0;u<nCampos;u++){

printf("Nombre del campo %d:\n",u+1);

metadatos[u]=strdynread();

}

return metadatos;

}

**redimEspacioTablas → tablas.h**

**La función recibe un apuntador doble a Tabla que es “conj”, el cual representa el espacio de tablas, y recibe un (int) m que es la nueva cantidad asignada de tablas que puede alojar el espacio de tablas.**

// Redimensionar/Cambiar tamaño de conjunto de tablas

Tabla \*\*redimEspacioTablas(Tabla \*\*conj,int m){

// Reasignamos el tamaño del espacio de tablas

conj=(Tabla \*\*)realloc(conj,m+1);

return conj;

}

**strdynread()**

**Función que permite leer cadenas de manera totalmente dinámica y sin un límite de caracteres. Lee una línea completa de entrada de datos (stdin) hasta que el usuario presione un salto de línea o defina un final de archivo (EOF, End Of File)**

char \*strdynread(){

// Declaramos la cadena que vamos a devolver en return al final, igual a NULL

char \*cad=NULL;

int ind=0,c=EOF;

// Seguir leyendo caracteres mientras c no sea c=0

while(c){

// Lee el carácter en cuestión

c=getc(stdin);

// Si este carácter c vale c=EOF o c es un salto de línea haz que sea c=0

if(c==EOF || c=='\n')

c=0;

// Cambiar el tamaño del bloque de memoria del apuntador cad en 1;

cad=realloc(cad,ind+1);

// Guardar el caracter c en cad[ind] con ind que es la posición actual de la cadena, y luego incrementar en 1 la posición actual

cad[ind++]=c;

// Al usar '++' con notación postfija en ind, primero se hace la asignación y luego se hace el incremento en ind

}

// Devolver la nueva cadena

return cad;

}