BBDD Clave-Valor

PRÁCTICA FUNDAMENTOS DE PROGRAMACIÓN

Se pide realizar un programa en C (no en C++), en línea de comandos, que emule el funcionamiento de un gestor de bases de datos **NoSQL** del tipo **Clave-Valor** (o "key-value"). Como se indica, se trata de emular una pequeña parte de la funcionalidad de este tipo de gestores de bases de datos, no de implementar uno completo o profesional.

Las bases de datos Clave-Valor se caracterizan por almacenar información utilizando pares <clave,valor>, donde la "clave" es un identificador único y el "valor" es un sentido o significado asociado a dicha clave de tal forma que aporta algún tipo de información sobre la misma, dicho "valor" podrá ser de alguno de los siguientes tipos:

- **numeric** (numérico entero o real)
- **string** (cadena de caracteres)
- **date** (fecha en formato YYYY-MM-DD o YYYY/MM/DD)
- **list** (lista de valores separados por el carácter 'ø' → código ASCII 248)

Adicionalmente, una base de datos Clave-Valor se organiza en "contenedores" (o "cabinets") que no son más que un conjunto de pares <clave,valor>. Dentro de un determinado cabinet las claves serán únicas y no podrán repetirse. Un ejemplo:

Base de Datos "usuarios"

Contenedor: password

Clave	Valor
jjgarcia	qwerty7
marta99	poiuy6
raulperez	1234
fecha	2017-10-21
v1	Hola Mundo
v2	comor?
v3	-5

Contenedor: nombre

Clave	Valor
jjgarcia	Juan José García
marta99	Marta López
raulperez	Raul Pérez
datos	d1ød2ød3

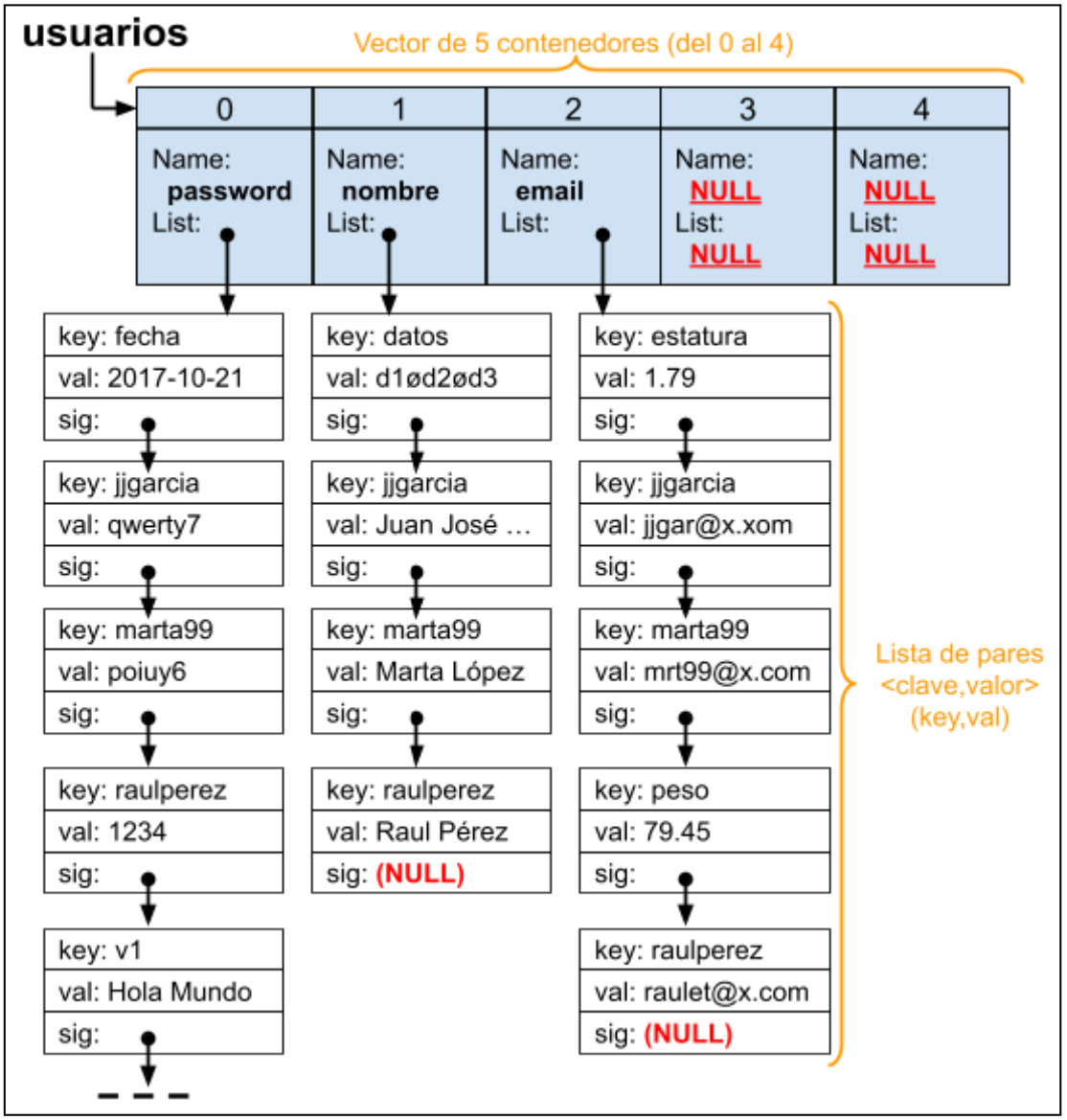
Contenedor: email

Clave	Valor
jjgarcia	jjgar@x.com
marta99	mrt99@x.com
raulperez	raulet@x.com
peso	79.45
estatura	1.79

En el ejemplo anterior se muestra una base de datos clave-valor llamada "usuarios" con tres contenedores o "cabinets" denominados 'password', 'nombre' y 'email' y una serie de elementos <clave,valor> en cada uno de dichos contenedores. Nótese que cada contenedor puede tener valores de tipos variados, los valores de un mismo contenedor no tienen por qué ser del mismo tipo. También se observa que algunas claves pueden estar

repetidas en distintos contenedores, pero no hay claves repetidas dentro del mismo contenedor. Igualmente, el número de elementos (pares <clave,valor>) dentro de un mismo contenedor será variable y podrá coincidir, o no, con el número de elementos de otros contenedores.

Aunque hayan datos de diferentes tipos, internamente, en memoria, todo se almacenará como una cadena de texto. Un par <clave,valor> serán dos cadenas de caracteres, una para la clave, y otra para el valor. Un contenedor será una lista enlazada y ordenada de pares <clave,valor>. Por último, una base de datos será un vector de contenedores, se establece como restricción en esta práctica que una base de datos podrá tener como máximo 5 contenedores, por tanto dicho vector tendrá 5 componentes. Gráficamente, la base de datos anterior se almacenará en memoria como indica la siguiente figura.



Para poder representar esta estructura de datos en un programa en C se utilizarán las siguientes definiciones de **struct**'s (estructuras en C):

```c
typedef struct KV // 'KV' abreviatura de Key-Value (clave-Valor)
{
    char *key;
    char *val;
    struct KV *next; // puntero a la siguiente estructura para la lista
} keyval;

typedef struct CAB // 'CAB' abreviatura de Cabinet (contenedor)
{
    char *name;    // nombre del contenedor
    keyval *list;  // puntero primer nodo a lista de pares <key,val>
    int len;       // longitud del contenedor (de la lista)
} cabinet;
```

Una vez definidas estas estructuras en C, una base de datos clave-valor de 5 contenedores de declarará del siguiente modo:

```c
cabinet bd[5];
```

La variable 'bd' (base de datos) es un vector de 5 componentes (del 0 al 4) cada uno de los cuales es una estructura de tipo 'cabinet'. Nótese que lo que han de ser cadenas de caracteres ('key', 'val', 'name') se han declarado de tipo puntero, el programa deberá reservar en cada momento la memoria justa necesaria para almacenar dichas cadenas de caracteres, y liberarla cuando corresponda (gestión de memoria dinámica).

Inicialmente, cuando arranca el programa, al no haber ninguna base de datos almacenada en memoria la variable 'bd' deberá ser inicializada con una función como esta:

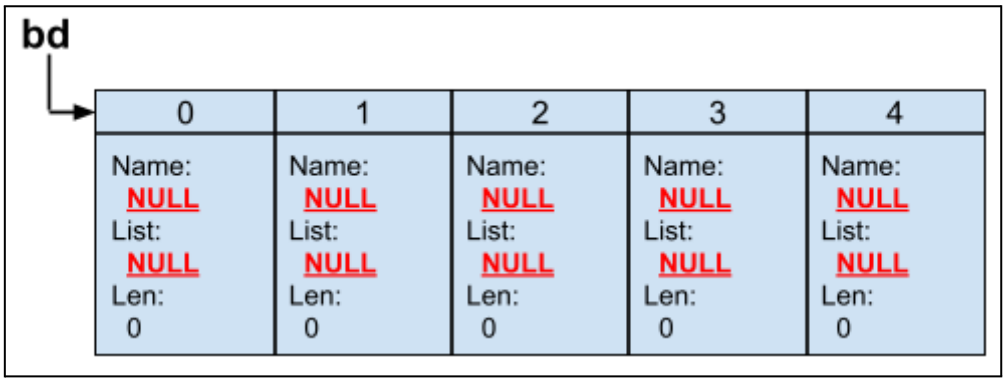
```c
void InicializarBD(cabinet *bd)
{
    int i;

    for(i=0 ; i<5 ; i++)
    {
        bd[i].name=NULL;
        bd[i].list=NULL;
        bd[i].len=0;
    }
}
```

Con esta función, y tras haber declarado la variable 'bd' (ver código más arriba), para inicializar la base de datos como "**vacía**", bastará con ejecutar la siguiente línea:

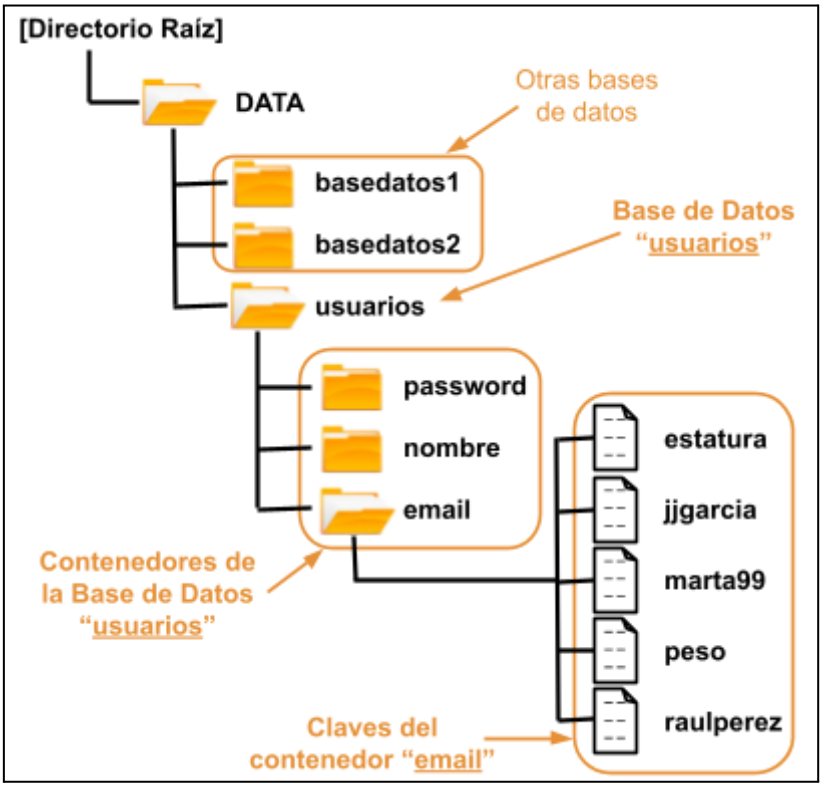
```c
InicializarBD(bd);
```

Tras lo cual, gráficamente, la base de datos quedaría del siguiente modo:



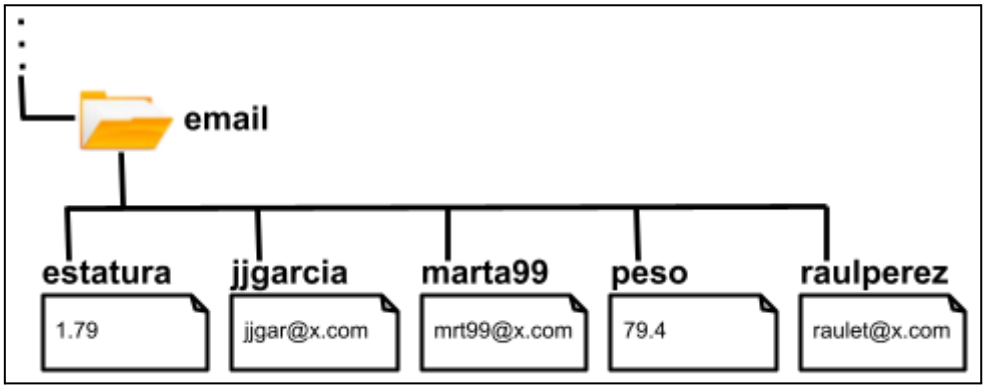
La base de datos ‘bd’ tendrá 5 contenedores o ‘cabinets’ vacíos y sin nombre.

Adicionalmente, una base de datos se deberá poder almacenar en disco para poder ser recuperada en otro momento (o para migrarla, intercambiarla o hacer una copia de seguridad). El directorio raíz donde se encuentre el programa ejecutable deberá tener una carpeta de nombre “DATA” y dentro de ella, en forma de árbol de directorios, deberán estar las bases de datos a las que se podrá acceder desde el programa. La base de datos “usuarios” del ejemplo anterior, almacenada en disco, creará la siguiente estructura de directorios y ficheros:

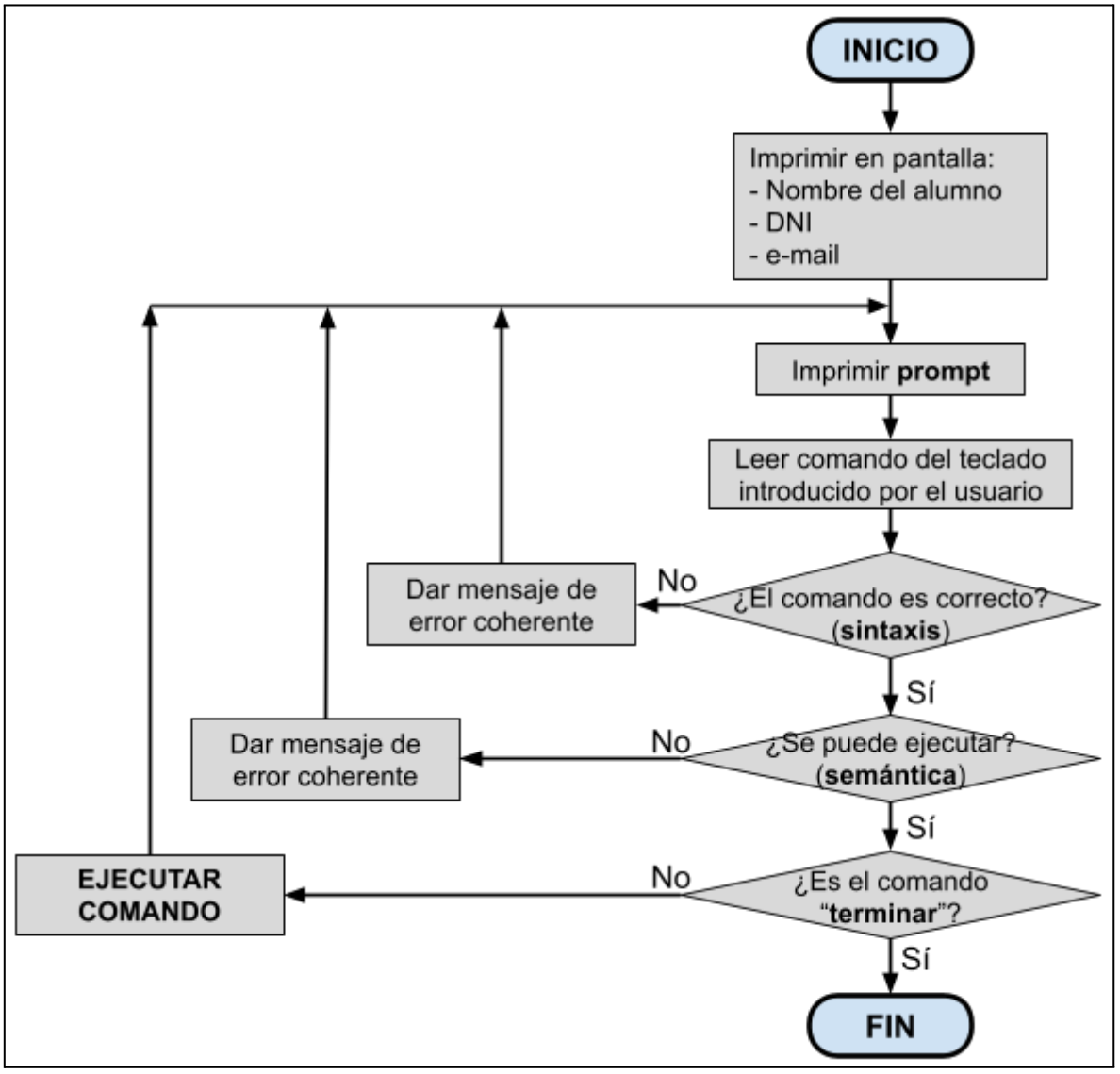


Cada base de datos estará en una carpeta o directorio dentro de la carpeta “DATA”. A su vez, cada contenedor será otra carpeta dentro de la base de datos. Por último, las claves

serán ficheros de texto homónimos dentro de sus respectivos contenedores, y cada uno de esos ficheros de texto tendrá como contenido escrito el valor asociado a la clave correspondiente en una única línea.



Vista la estructura de una base datos clave-valor, tanto en memoria como en disco, pasamos a describir cómo debe ser el funcionamiento de la práctica a realizar. De forma general, el programa que se pide deberá obedecer al siguiente diagrama de flujo:



El **prompt** es un elemento característico de las aplicaciones de consola en las que el usuario introduce comandos por teclado para interactuar con el programa que se está ejecutando. Te trata un texto corto que precede al cursor en el punto en que el usuario debe introducir dichos comandos. En el caso de esta práctica, el prompt será un texto que aporte cierta información sobre el estado en el que se encuentra el programa. Al arrancar el programa el prompt deberá tener esta forma que llamaremos '**prompt-1**':

```
[.]>>
```

Cuando hay una base de datos activa el prompt deberá indicar el nombre de dicha base de datos entre corchetes, llamaremos a esto '**prompt-2**':

```
[usuarios]>>
```

Por último, cuando la base de datos se modifica, bien porque se agrega un nuevo dato, bien porque se elimina o actualiza, pasaremos al '**prompt-3**' en el que se indica con un asterisco que hay cambios no guardados:

```
[usuarios]*>>
```

Cuando estando en el '**prompt-3**' se ejecuta una orden de guardar los cambios, se pasaría al '**prompt-2**'. Si se carga en memoria una nueva base de datos, entonces se pasa también al '**prompt-2**', pero cambiando el nombre de la base de datos por el de la nueva que se acaba de cargar.

En cualquiera de los casos, siempre que se imprime el prompt en la consola, justo a la derecha del mismo debe aparecer el cursor para que el usuario pueda introducir una orden o comando que posteriormente la aplicación analizará y, si es posible, ejecutará. Para leer dichos comandos se usará la función de C `gets(...)`.

Tras introducir un comando (y pulsar [INTRO]) el programa debe comprobar si dicho comando está correctamente construido y si se puede ejecutar, es decir, deberá analizarse sintáctica y semánticamente. En caso de que se detecte algún error o de que la orden no se pueda ejecutar, el programa deberá imprimir en pantalla un mensaje indicando el error que se ha cometido para inmediatamente después, sin hacer ninguna pausa, volver a mostrar el prompt y permitir que el usuario introduzca otro comando.

Cuando el comando es sintáctica y semánticamente válido, en caso de que se trate del comando que termina la aplicación, se finalizará la misma; si se trata de cualquier otra orden, se ejecutará convenientemente, realizando las operaciones que corresponda según el comando de que se trate, tras lo cual, sin hacer ninguna pausa, se volverá a mostrar el prompt para que el usuario pueda introducir una nueva orden.

A continuación, se describen todos los comando que deberá reconocer y ejecutar la aplicación, en cada caso los comandos serán correctos si incluyen los parámetros que en cada caso se indique. Los comandos se escribirán en '**negrita**' y los parámetros en 'texto normal'. Cuando un comando o grupo de comandos aparezca entre '[corchetes]', eso querrá decir que se trata de un parámetro opcional; cuando aparezcan puntos suspensivos '. . .' se estará indicando que se trata de un número indefinido de parámetros.

COMANDOS:

quit

Este comando es el que termina la aplicación, no tiene ningún argumento (en caso de tenerlo sería erróneo y no se ejecutaría). Lo único que hace es liberar la memoria que hubiera ocupada y finalizar la ejecución del programa.

Comandos para la base de dados (todos incorporan el sufijo 'db' → 'database'):

newdb nombre

Crea una nueva base de datos con el nombre indicado en el único parámetro que debe tener este comando. La nueva base de datos creada, que por defecto estará vacía, se convertirá en la base de datos activa y se cambiará el prompt con su nombre. Dicha base de datos estará tan solo en memoria (se podrá almacenar en disco después con el siguiente comando), al no estar guardada en disco el prompt deberá mostrar el asterisco inicialmente (prompt-3). La memoria de la base de datos que estuviera activa anteriormente deberá ser liberada debidamente.

savedb

Comando para guardar en disco la base de datos actual, se crearán las carpetas y ficheros que sean necesarios. Si la base de datos ya existía en el disco se actualizarán las carpetas y ficheros correspondientes (una forma podría ser borrar todas las carpetas y ficheros antiguos y volverlos a crearlos de nuevo con los datos actualizados). Después de ejecutar este comando el prompt debe quedar sin el asterisco (prompt-2).

listdb

Muestra en la consola la lista de bases de datos disponibles en líneas consecutivas, es decir, muestra los nombres de todas las carpetas que hay dentro del directorio "DATA".

activedb nombre

Pone la base de datos indicada por el parámetro 'nombre' como base de datos activa, cargando los datos de dicha base de datos en memoria (y liberando la memoria de la base de datos hubiera anteriormente activa). Cambia el prompt con el nombre de la nueva base de datos activada que, al estar recién cargada y sin cambios, no mostrará el asterisco (prompt-2). Obviamente, para activar una base

de datos esta debe existir en el disco (dentro del directorio 'DATA'), si la base de datos no existe se indicará con un mensaje de error y se quedará como base de datos activa la que hubiera en el momento previo.

Comandos para los contenedores (todos incorporan el sufijo 'cab' → 'cabinet'):

Para que se puedan ejecutar estos comandos es IMPRESCINDIBLE que haya una base de datos activa, si no la hubiera no se podrán ejecutar y el programa deberá responder con un mensaje de error que indique la no existencia de dicha base de datos activa. Ninguno de estos comandos afectará a las carpetas o ficheros que haya guardados en disco, solo actuarán sobre la estructura de datos que hay en memoria.

newcab nombre

Crea un nuevo contenedor con el nombre indicado que en principio estará vacío. Como cada base de datos no puede tener más de 5 contenedores, si ya los hubiera, el nuevo contenedor no se podrá crear, por lo que se responderá con el correspondiente mensaje de error indicándolo.

listcab

Muestra en la consola el listado de los contenedores de la base de datos activa, para cada contenedor mostrará, en una línea por cada contenedor, su nombre y el número de elementos (pares <clave,valor>) que contenga. Los contenedores aún no definidos, si los hubiera, no se mostrarán.

activecab nombre

Pone el contenedor indicado en el parámetro 'nombre' como cabinet activo, el comando responderá con un mensaje de confirmación similar a "cabinet 'nombre' activado". En el caso de que no exista un contenedor con el nombre indicado el programa responderá con un mensaje similar a "'nombre' no encontrado" y seguirá como activo el contenedor que hubiera antes de ejecutar el comando.

Comandos para los elementos <clave,valor>:

Para que se puedan ejecutar estos comandos es IMPRESCINDIBLE que haya una base de datos y un contenedor activos, si no los hubiera no se podrán ejecutar y el programa deberá responder con un mensaje de error avisando de dicho error. Ninguno de estos comandos afectará a las carpetas o ficheros que haya guardados en disco, solo actuarán sobre la estructura de datos que hay en memoria.

set key value

Crea o actualiza en el contenedor activo la clave 'key' con el valor 'value'. Si la clave 'key' ya existe se actualiza el valor, sino se crea nueva con el valor indicado. Recuérdese que la lista de claves (el contenedor) debe estar ordenada lexicográficamente por el nombre de la clave ('key').

get key

Muestra en pantalla el valor de la clave ‘key’ del contenedor activo. Si la clave no existe se indicará con un mensaje de error. Si la clave hace referencia a un elemento de tipo ‘**list**’ (lista de valores) el programa responderá con un mensaje similar a “‘key’ es una lista” (para ver los elementos de una lista ver comando ‘range’ más abajo).

del key

Elimina del contenedor activo la clave ‘key’, el programa responderá con un mensaje de confirmación de que la clave se ha borrado, pero si la clave no existe responderá con un mensaje de error.

rpush key value

(Right Push). Añade por la derecha al valor de la clave ‘key’ el valor ‘value’, convierte a dicha clave en un dato de tipo ‘**lista**’ con sus elementos separados por el código ASCII 248 ('ø'). La clave debe existir, en caso contrario se indicará con un mensaje de error.

lpush key value

(Left Push). Análogo al anterior pero añadiendo el valor por la izquierda.

rpop key

(Right Pop). Operación contraria a ‘rpush’, elimina un elemento por la derecha de la lista de elementos cuya clave es ‘key’. Es necesario que el elemento ‘key’ exista y que sea una lista (más de un valor), si no es así el programa devolverá el mensaje de error que corresponda.

lpop key

(Left Pop). Análogo al anterior pero eliminando el valor por la izquierda.

range key [i j]

Los parámetros ‘i’ y ‘j’ son opcionales, sin ellos se mostrará en la consola todos los valores de la lista precedidos de su índice contando desde cero en adelante. Los parámetros ‘i’ y ‘j’ deben ser números enteros positivos y se debe cumplir que ‘i’ es menor o igual que ‘j’. Si se incluyen estos parámetros en el comando se mostrarán por pantalla solo los elementos desde la ‘i’ hasta la ‘j’. Obviamente, tanto ‘i’ como ‘j’ deberán estar dentro del rango de valores de la lista (p.e.: no se puede ver el elemento 8 de una lista que solo tiene 5 valores). Si no se dan todas las condiciones para que el comando se pueda ejecutar el programa responderá con el mensaje de error que corresponda.

key pattern

Este comando busca claves en el contenedor activo en función del patrón ‘pattern’ especificado. Este patrón deberá indicar el nombre de una clave o bien una parte

del nombre de una clave acompañada del comodín '*' (asterisco), que podrá ir al principio del patrón, al final o al principio y final, es decir, el patrón podrá tener alguna de las siguientes formas:

- `nombre` → devuelve la clave 'nombre'
- `nom*` → devuelve todas las clave que empiecen por 'nom'
- `*nom` → devuelve todas las clave que terminen por 'nom'
- `*nom*` → devuelve todas las clave que contengan 'nom'
- `*` → devuelve todas las claves del contenedor

En consola se mostrará el nombre de cada clave devuelta seguida de su valor, excepto en el caso de que la clave devuelta sea de tipo '**list**' en cuyo caso, en vez del valor, se mostrará el texto '<LIST>'.

`inc` `key`

Incrementa en una unidad el valor de la clave 'key'. Esta operación solo puede ejecutarse sobre valores de tipo '**numeric**' o '**date**'. En el caso de valores numéricos suma 1 al valor y en el caso de valores de tipo fecha, esta se incrementará en un día dando lugar a una nueva fecha correcta (validación de fechas). Si se aplica este comando a valores de tipo '**string**' o '**list**', no se podrá ejecutar y el programa devolverá un mensaje de error indicándolo.

`dec` `key`

Análogo al anterior pero decrementando o restando 1 al valor.

Y hasta aquí la lista de comandos que el programa debe reconocer y ejecutar, en todos los casos, si el comando fuera incorrecto (o fuera un comando no descrito en este enunciado), o si los parámetros que acompañan al comando no fueran los adecuados el programa siempre responderá con un mensaje de error que informe al usuario de cuál ha sido el error cometido al escribir la orden.

Todos los comandos que modifican algo en la base de datos forzarán a que en el prompt de la consola de comandos aparezca el asterisco que indica que la base de datos tiene cambios sin guardar (prompt-3). Solo cuando se guardan esos cambios en disco (**`savedb`**) o cuando se cambia la base de datos activa (**`activedb`**) se eliminará el asterisco del prompt (prompt-2).

MUY IMPORTANTE: en todo momento el programa deberá hacer una correcta gestión de memoria dinámica, sin dejar basura en memoria. Se tendrán que ejecutar las operaciones de reserva y liberación de memoria que corresponda para este fin.

Una dificultad que tendrá la aplicación es que a la hora de introducir comandos por teclado se permitirá que haya tokens que contengan espacios en blanco siempre que se

hayan escrito entre comillas, para entender esto vamos a suponer que tenemos una determinada base de datos y un contenedor activos sobre los que se van a ejecutar los siguientes comandos de ejemplo:

set nom1 Luis

CORRECTO → Almacena en el contenedor activo el par <nom1,Luis>.

set nom2 Luis Perez

INCORRECTO → El comando 'set' debe ir seguido de dos parámetros, la 'clave' y el 'valor', pero en este ejemplo hay un tercer parámetro ('Perez') por lo que el programa, en este caso, debería responder con un mensaje de error similar a "Número de parámetros incorrecto".

set nom2 "Luis Perez"

CORRECTO → Todo el texto que va entre comillas se considera una única cadena y por tanto, un único parámetro. En este caso se almacenará en el contenedor activo el par <nom2,Luis Perez> (sin las comillas).

En general, siempre que el usuario escriba un comando por teclado, cuando haya parte del texto entre comillas, todo ese texto entrecomillado se considerará un único parámetro. Parámetros como los siguientes serían correctos:

set "user 1" "Luis Perez Lopez"

CORRECTO → Crea el par <user 1,Luis Perez Lopez>.

newdb "mis usuarios"

CORRECTO → Crea una nueva base de datos llamada "mis usuarios".

newcab "Alumnos FP"

CORRECTO → Crea un contenedor de nombre "Alumnos FP".

Naturalmente, las comillas, cuando las haya, deberán ir por pares y tener texto entre ellas, de lo contrario el comando también sería incorrecto. Ejemplos:

set user "Luis Perez

INCORRECTO → Abre comillas, pero no las cierra.

newdb " "

INCORRECTO → la cadena entre comillas está en blanco.

Ampliación 1:
Examen Diciembre 2017

Ejercicio 1 (2.5 puntos)

Modificar el prompt para que también refleje el nombre del contenedor activo.

```
prompt-1 → [./.]>>
prompt-2 → [DB/.]>>     ||     [DB/Cab]>>
prompt-3 → [DB/.]*>>    ||     [DB/Cab]*>>
```

Ejercicio 2 (2.5 puntos)

Modificar comando "**listcab**" para que muestre además, junto al nombre de cada cabinet el tamaño que ocupan en memoria las claves y los valores que contiene, dicho tamaño deberá ser la suma de todos los caracteres de todas esas claves y valores.

Ejercicio 3 (2.5 puntos)

Modificar el comando "**set**" para que admita la siguiente sintaxis:

```
set key value [key value ...]
```

es decir, un número indefinido de pares <clave/valor> para añadir al cabinet o modificarlos (hasta un máximo de 10 pares).

Ejercicio 4 (2.5 puntos)

Nuevo comando "**sort key**" para ordenar lexicográficamente el contenido de una clave de tipo "list". Si la clave no existe o no es de tipo "list" → error. Este comando, cuando no se produce error, no muestra nada en pantalla.

Avisos:

- La duración del examen es de 3 horas
- Si la práctica contiene algún error, este podría restar a la nota final del examen
- **MUY IMPORTANTE:** INDICAR AL PRINCIPIO DEL PROGRAMA QUE EJERCICIOS ESTÁN HECHOS Y CUÁLES NO (no hacerlo restará 1 punto)

```
- Ejercicio 1: HECHO / SIN HACER
- Ejercicio 2: . . .
```

Ampliación 2:
Examen Enero 2018

Ejercicio 1 (2.5 puntos)
Modificar el prompt para que en vez de un asterisco aparezca en su lugar un contador de los cambios no guardados. Cada vez que se guarde la base de datos el contador deberá inicializarse a cero.

Ejercicio 2 (2.5 puntos)
Crea un nuevo comando "**rnkey**" (rename-key) para cambiar el nombre de una clave:

```
rnkey keyActual keyNueva
```

Se busca la clave "keyActual" en el cabinet activo y se cambia dicho nombre por "keyNueva" (se debe hacer una gestión correcta de la memoria dinámica).

Ejercicio 3 (2.5 puntos)
Modificar los comandos "**lpush**" y "**rpush**" para que admitan la siguiente sintaxis:

```
lpush key val [val2 ...]
rpush key val [val2 ...]
```

es decir, un número indefinido de valores para añadir a la lista bien por la izquierda o bien por la derecha, según corresponda (hasta un máximo de 10 valores).

Ejercicio 4 (2.5 puntos)
Modificar el comando "**sort**" para que admita de manera opcional un parámetro que indique si la lista debe ser ordenada ascendente o descendentemente:

```
sort key [asc/des]
```

Avisos:

- La duración del examen es de 3 horas
- Si la práctica contiene algún error, este podría restar a la nota final del examen
- **MUY IMPORTANTE:** INDICAR AL PRINCIPIO DEL PROGRAMA QUE EJERCICIOS ESTÁN HECHOS Y CUÁLES NO **(no hacerlo restará 1 punto)**

```
- Ejercicio 1: HECHO / SIN HACER
- Ejercicio 2: . . .
```

Ampliación 3:
Examen Septiembre 2018

Ejercicio 1 (2.5 puntos)
Verificar si hay cambios no guardados antes de los comandos **`quit`**, **`newdb`** y **`activedb`**. En caso de que haya cambios sin guardar el programa pedirá una confirmación del tipo:

```
Los cambios no guardados se perderan, Desea continuar? (si/no):
```

Ejercicio 2 (2.5 puntos)
Modificar los comandos **`inc`** y **`dec`** para que admita un número opcional (**`[n]`**) que podrá ser entero o real, positivo o negativo y que se sumará o restará a la clave correspondiente. En caso de que trate de una fecha se sumará la parte entera ignorando los decimales.

Ejercicio 3 (2.5 puntos)
Modificar el comando **`del`** para que admita de forma opcional múltiples claves a borrar (hasta un máximo de 10 claves), es decir, que siga la sintaxis:

```
del key [key2 ...]
```

El comando ignorará si alguna de las claves indicadas no existe y devolverá un mensaje por pantalla indicando el número de claves borradas.

Ejercicio 4 (2.5 puntos)
Nuevo comando **`copycab namedb`**, copia el cabinet activo en la base de datos 'namedb' en el disco. Debe haber un cabinet activo, la base de datos 'namedb' tiene que existir en el disco, no debe tener un cabinet con el mismo nombre, y al menos un hueco para el nuevo cabinet (menos de 5). En caso de que el comando no pueda ejecutarse el programa responderá con un mensaje indicando el motivo.

Avisos:

- La duración del examen es de 3 horas
- Si la práctica contiene algún error, este podría restar a la nota final del examen
- **MUY IMPORTANTE:** INDICAR AL PRINCIPIO DEL PROGRAMA QUE EJERCICIOS ESTÁN HECHOS Y CUÁLES NO **(no hacerlo restará 1 punto)**

```
- Ejercicio 1: HECHO / SIN HACER
- Ejercicio 2: . . .
```