Estructura de Datos, Práctica 3: Memoria

Leandro García y Fabián Gutiérrez (Pareja 07)

17 de diciembre de 2019

1. Tablas y tipos de datos

En primer lugar, se definió la estructura de tabla de la siguiente forma:

```
struct table_ {
    FILE *f;
    int ncols;
    type_t *types;
    char **reg;
    long first_pos;
    long last_pos;
};
```

Donde *last_pos* se refiere a la próxima posición de fichero en la que insertar un nuevo registro, y *reg* almacena cada dato (literal) del último registro leído con *table_read_record*.

En cuanto a la estructura del fichero, se opta por la cabecera sugerida (número de columnas y el tipo de dato de cada una), pero para los registros, en lugar de indicar la longitud total, se indica la longitud de cada dato, y luego el dato en sí. Esto complica (ligeramente) la lectura, que con la alternativa sugerida en el enunciado consistiría en copiar el registro entero (conocida su longitud) en una cadena de caracteres de la estructura *table_t*. Pero facilita la escritura, ya que no hay que calcular la longitud total del registro, sino que se va escribiendo de forma secuencial; y facilita el acceso a una columna dada (por ejemplo, la i-ésima), ya que no hay que calcular el desfase del i-ésimo dato en la cadena de caracteres leída, si no que se accede directamente a la posición i-ésima del arreglo de cadenas *reg*.

Por otra parte, implementar los nuevos tipos de datos se realizó tomando como referencia los tipos ya definidos e implementados en *types.h* y *types.c*. Además, se creó un programa de prueba que verifica el correcto funcionamiento de cada función con cada tipo de dato, indicando, en caso de error, qué función falla.

```
fabian@DESKTOP-TDV6L08:/mnt/c/NetBeans/EDAT_P3/development$ ./type_test.exe
4
prueba
123456789
1.002000
Salida correcta.
```

Figura 1: Salida del programa de prueba *type_test*

Finalmente, dado que los ficheros suministrados al inicio de la práctica tenían errores y, por ende, las pruebas realizadas al módulo *table* fracasaban (posteriormente se reemplazaron estos por

los nuevos ficheros depurados y se superaron las pruebas [Figura 3]), se creó otro programa de prueba que verificaba el correcto funcionamiento de la lectura y escritura de tablas, además de los tipos de datos ya que la tabla de prueba utilizada contenía columnas de cada tipo.

```
fabian@DESKTOP-TDV6L08:/mnt/c/NetBeans/P3_EDAT_G1201_P07/development$ ./table_test.exe
Ryan  Gosling  39  111438  1.840000
Emma  Stone  31  1297015  1.680000
```

Figura 2: Salida (correcta) del programa de prueba *table_ test*

```
e402064@8B-26-19-26:~/UnidadH/EDAT_P3/install$ ./test
Text: none, table: none, index: none,
$ verify test.txt
1       Hello world     10          text file
1       Hello world     10          table
2       Hello world     11          text file
2       Hello world     11          table
3       Hello world     11          text file
3       Hello world     11          table
4       Hello world     11          text file
4       Hello world     11          table
5       Hello world     11          text file
5       Hello world     11          table
```

Figura 3: Salida del programa de prueba *test* suministrado

2. Índices

Para los índices se definieron las siguientes estructuras:

```
typedef struct {
    int key;        /*It only works with integers by now*/
    int n_offsets;
    long *offsets;
} entry;

struct index_ {
    FILE *f;
    type_t type;
    int n_keys;
    entry **entries;
};
```

Donde *entry* es un tipo abstracto de datos que almacena la información asociada a una entrada del índice, y la estructura *index_ t* almacena un arreglo de estos.

En cuanto a la estructura del fichero donde se almacena el índice, se usa de nuevo la cabecera sugerida (tipo y número de claves) y para cada entrada se indica la clave, el número de registros asociados a dicha clave y los offsets de cada uno de estos (esto es, las direcciones en bytes en las que se encuentran los registros mencionados).

También cabe destacar que se hace una modificación a la función de búsqueda binaria propuesta, precisamente en el retorno. El retorno propuesto para el caso en que no se encuentra la clave es (siguiendo los nombres de variables del enunciado) $-(\text{middle}+1)$. Sin embargo, esta implementación no diferencia entre el caso en el que la clave a buscar gana en la última comparación y el caso en el que pierde; por ello se optó por devolver $-(\text{middle} + 1 + \text{flag})$ donde *flag* es un entero que vale 1 si la clave gana la última comparación y 0 si pierde. De este modo, siendo r el retorno de la búsqueda binaria, $-(r + 1)$ es la posición en la que se ha de insertar la nueva clave.

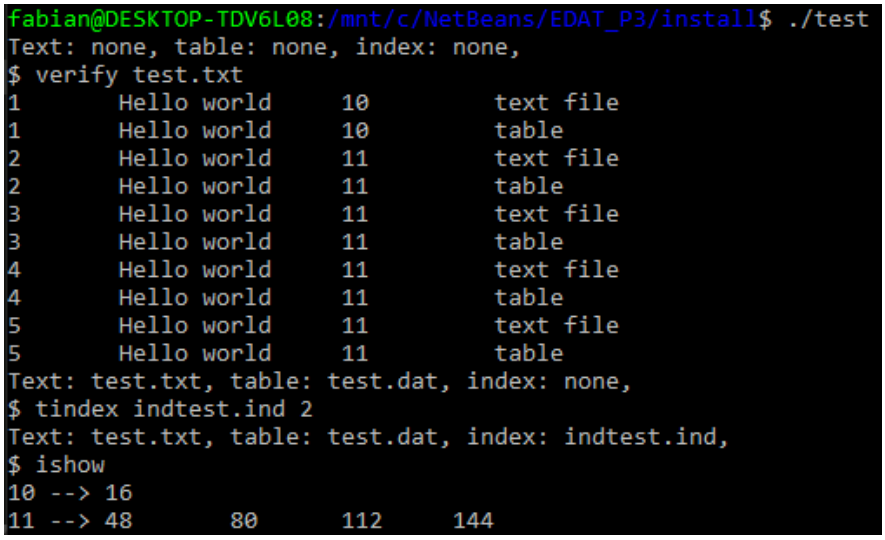

Figura 4: Salida del programa de prueba *test* suministrado (con índices)