Trabajo Práctico 1
Tabla de Traducciones

Organización del Computador 2

Primer Cuatrimestre 2016

1. Introducción

Este trabajo practico consiste en implementar una tabla de traducciones. Es decir, una estructura de datos que permite dado un valor, transformarlo en otro. Las diferentes funciones a implementar permiten crear/borrar una tabla de traducciones, agregar/borrar traducciones, así como también realizar un conjunto adicional de operaciones más complejas.

La implementación de la tabla de traducciones considera algunas restricciones. Debe ser una tabla que transforme cualquier valor de 3 bytes en un valor cualquiera de 15 bytes. De guardar toda la tabla en memoria, ésta ocuparía alrededor de 256MB de memoria. Considerando que no todas las traducciones posibles serán almacenadas en la tabla, se propone una estructura alternativa a fin de ahorrar espacio. Dicha estructura es explicada a continuación.

1.1. Tipo Tabla de Traducciones

La estructura de una tabla de traducciones esta compuesta por un bloque de memoria denominado `tdt` que contiene un puntero a una subtabla, un texto de identificación y la cantidad de traducciones que tiene la tabla. A su vez, la subtabla apuntada, denominada `tdtN1`, será la tabla de traducciones de primer nivel, ésta contiene 256 entradas donde cada una apunta a una nueva subtabla de nivel 2 (`tdtN2`) con las mismas características que la primera. La tabla de nivel 2 apunta entonces a una nueva tabla denominada `tdtN3` que contiene las traducciones correspondientes. En la figura 1, se puede ver un ejemplo esquemático de como están relacionadas las diferentes subtablas que componen al tipo `tabla de traducciones`.

```
typedef struct valor_t {
  uint8_t val[15];
} __attribute__((__packed__)) valor;

typedef struct clave_t {
  uint8_t cla[3];
} __attribute__((__packed__)) clave;

typedef struct valorValido_t {
  struct valor_t valor;
  uint8_t valido;
} __attribute__((__packed__)) valorValido;

typedef struct tdtN3_t {
  struct valorValido_t entradas[256];
} __attribute__((__packed__)) tdtN3;
```

```
typedef struct tdtN2_t {
  struct tdtN3_t* entradas[256];
} __attribute__((__packed__)) tdtN2;

typedef struct tdtN1_t {
  struct tdtN2_t* entradas[256];
} __attribute__((__packed__)) tdtN1;

typedef struct tdt_t {
  char* indentificacion;
  struct tdtN1_t* primera;
  uint32_t cantidad;
} __attribute__((__packed__)) tdt;
```

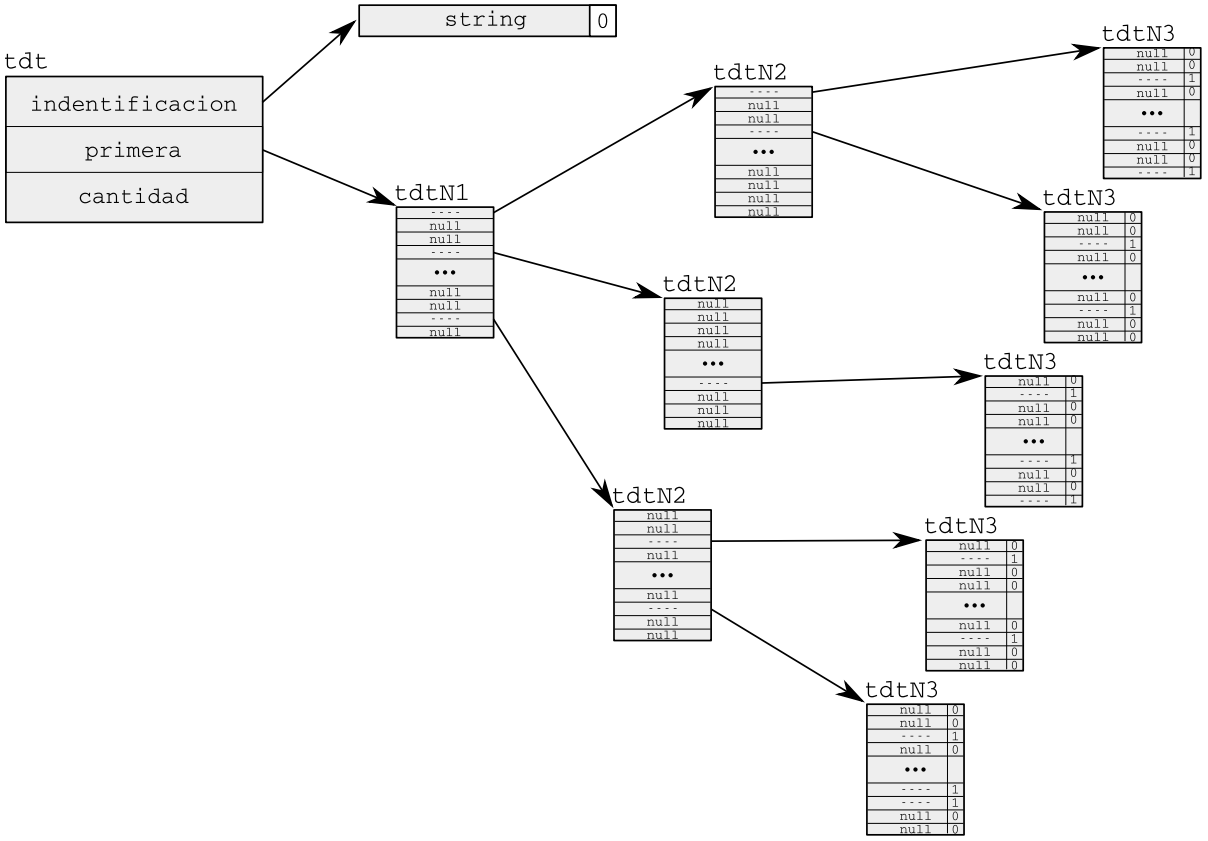

Figura 1: Ejemplo de una estructura tabla de traducciones

Para realizar una traducción, primero se descompone la clave en tres partes de un byte cada una. Estos valores serán los índices en las subtablas, siendo el byte mas significativo el correspondiente al índice de la subtabla de nivel 1 y los restantes para el resto de las subtablas en orden.

Una traducción se considerará válida si el valor `valido` dentro de la tabla de tercer nivel correspondiente a dicha traducción es distinto de cero.

Inicialmente una tabla de traducciones no tiene ninguna subtabla, ya que se debe mantener el invariante. **Invariante** toda subtabla apunta al menos una subtabla o tiene un dato válido.

En la figura 2 se encuentra un ejemplo completo de una tabla con 4 traducciones válidas.

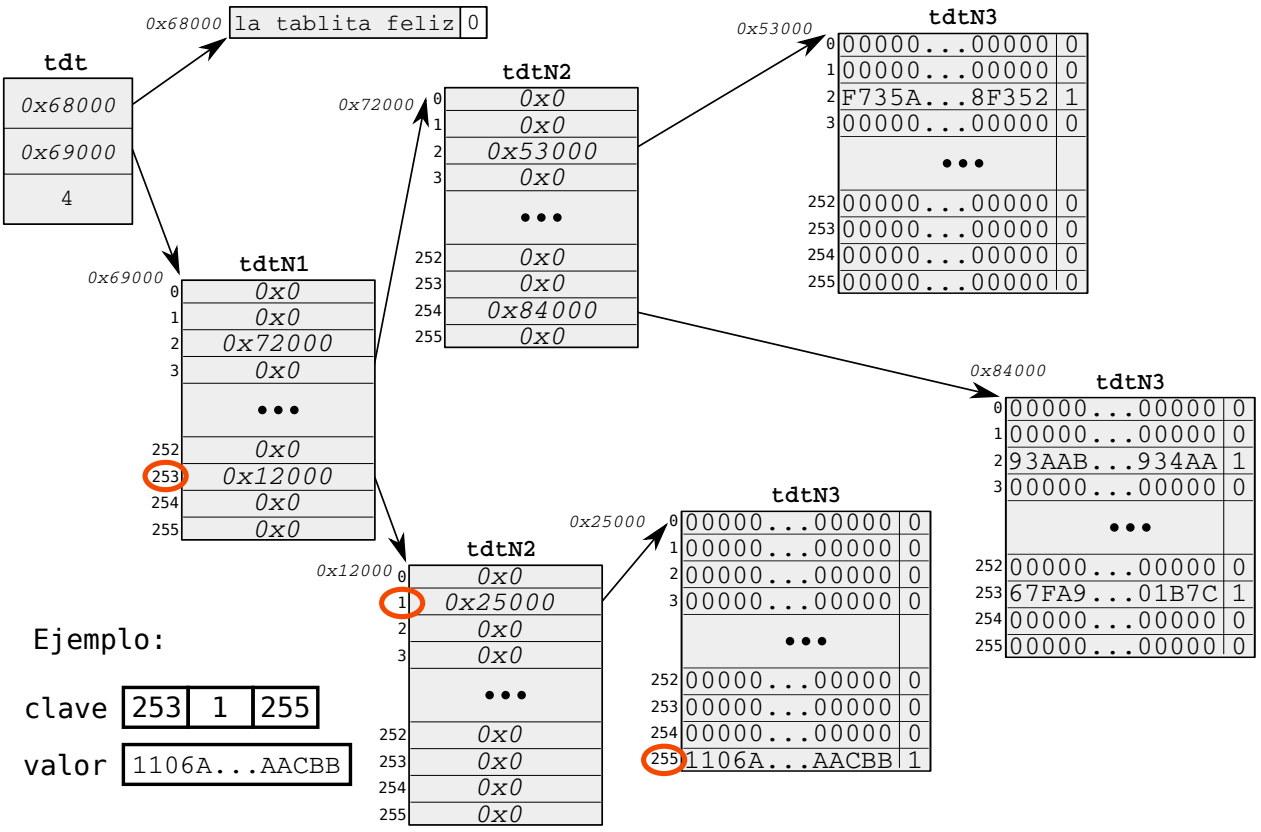

Figura 2: Ejemplo de una instancia de tabla de traducciones. En circulo se descompone una traducción de ejemplo.

Otras estructuras

Además del tipo `tabla de traducciones`, se deben considerar dos estructuras útiles para almacenar información sobre traducciones. La estructura `bloque` contiene tanto una clave como un valor para traducir, agregar o borrar según corresponda. Por otro lado, la estructura `maxmin` contiene los valores máximos y mínimos de claves como valores.

```
typedef struct bloque_t {
  uint8_t clave[3];
  uint8_t valor[15];
} __attribute__((__packed__)) bloque;

typedef struct maxmin_t {
  uint8_t max_clave[3];
  uint8_t min_clave[3];
  uint8_t max_valor[15];
  uint8_t min_valor[15];
} __attribute__((__packed__)) maxmin;
```

Funciones de Tabla de Traducciones

- `tdt* tdt_crear(char* indentificacion);`
  Crea una tabla de traducciones vacía, es decir sin ninguna subtabla ni traducción. Solamente debe contener una copia de la identificación.

- `uint32_t tdt_cantidad(tdt* tabla);`
  Lee la cantidad de traducciones de una tabla.

- `void tdt_agregar(tdt* tabla, uint8_t* clave, uint8_t* valor);`
  Inserta una traducción respetando el procedimiento de la sección 1.1. De ser necesario crea las subtablas que se requieran.

- `void tdt_borrar(tdt* tabla, uint8_t* clave);`
  Borra una traducción respetando el procedimiento de la sección 1.1. Se deben borrar todas las tablas que no contengan traducciones.

- `void tdt_traducir(tdt* tabla, uint8_t* clave, uint8_t* valor);`
  Traduce una clave según la tabla. De no existir la traducción no se deberá modificar el parámetro *valor*.

- `void tdt_destruir(tdt** tabla);`
  Destruye todas las estructuras de una tabla de traducciones.

- `void tdt_imprimirTraducciones(tdt* tabla, char* file);`
  Imprime una lista de traducciones. Para esto se debe concatenar al archivo pasado como parámetro la lista de traducciones, con el siguiente formato.
  La primera linea deberá contener el identificador respetando el formato `"- $identificacion -"`. Las siguientes lineas contendrán en orden las traducciones una a una respetando el formato `"$clave => $valor"` para cada traducción (los valores numéricos deben imprimirse en hexadecimal de tamaño fijo). Ejemplo para la tabla de la figura 1, imprime:

  ```
  - la tablita feliz -
  020202 => F735A983485825923592340538F352
  02FE02 => 93AAB85672AB5832D485EE238934AA
  02FEFD => 67FA984B3875A98465D8375FF01B7C
  FD01FF => 1106AF39875A9845B48C38471AACBB
  ```

  (los valores fueron completados a 15 bytes, en la figura 1 se muestran resumidos)

Funciones avanzadas sobre Tabla de Traducciones

- `void tdt_recrear(tdt** tabla, char* indentificacion);`
  Dada una tabla, destruye su contenido y genera una nueva tabla vacía en su lugar. Setea su nuevo identificador; de ser este *null*, deja el identificador anterior.

- `void tdt_agregarBloque(tdt* tabla, bloque* b);`
  `void tdt_borrarBloque(tdt* tabla, bloque* b);`
  `void tdt_traducirBloque(tdt* tabla, bloque* b);`
  Operaciones en bloque. Realizan lo mismo que las operaciones de la sección anterior solo que tomando como entrada la *clave* y el *valor* de una estructura de *bloque*.

- `void tdt_agregarBloques(tdt* tabla, bloque** b);`
  `void tdt_borrarBloques(tdt* tabla, bloque** b);`
  `void tdt_traducirBloques(tdt* tabla, bloque** b);`
  Operaciones en cadena de bloques. Realizan lo mismo que las operaciones de la sección anterior pero sobre una cadena de punteros a bloques terminados en *null*.

- `maxmin* tdt_obtenerMaxMin(tdt* tabla);`
  Completa la estructura *maxmin* con el contenido de la máxima y mínima clave, y del máximo y mínimo valor de toda la tabla. Debe iterar todas las traducciones de la tabla y considerar el orden lexicográfico para la comparación.

2. Enunciado

Ejercicio 1

Implementar todas las funciones de **Tabla de traducciones** mencionadas anteriormente según se indica a continuación:

En lenguaje assembler

- `tdt* tdt_crear(char* indentificacion)` *(24 lineas)*
- `void tdt_recrear(tdt** tabla, char* indentificacion)` *(19 lineas)*
- `uint32_t tdt_cantidad(tdt* tabla)` *(2 lineas)*
- `void tdt_agregarBloque(tdt* tabla, bloque* b)` *(2 lineas)*
- `void tdt_borrarBloque(tdt* tabla, bloque* b)` *(1 lineas)*
- `void tdt_traducirBloque(tdt* tabla, bloque* b)` *(2 lineas)*
- `void tdt_agregarBloques(tdt* tabla, bloque** b)` *(14 lineas)*
- `void tdt_borrarBloques(tdt* tabla, bloque** b)` *(16 lineas)*
- `void tdt_traducirBloques(tdt* tabla, bloque** b)` *(14 lineas)*
- `void tdt_traducir(tdt* tabla, uint8_t* clave, uint8_t* valor)` *(26 lineas)*
- `void tdt_destruir(tdt** tabla)` *(51 lineas)*

En lenguaje C

- `void tdt_agregar(tdt* tabla, uint8_t* clave, uint8_t* valor)` *(20 lineas)*
- `void tdt_borrar(tdt* tabla, uint8_t* clave)` *(27 lineas)*
- `void tdt_imprimirTraducciones(tdt* tabla, char* file)` *(20 lineas)*
- `maxmin* tdt_obtenerMaxMin(tdt* tabla)` *(42 lineas)*

Entre paréntesis se menciona de forma ilustrativa, la cantidad de lineas que tomo cada función en la solución de la cátedra.

Ejercicio 2

Construir un programa de prueba (modificando `main.c`) que realice las siguientes acciones llamando a las funciones implementadas anteriormente:

1- Crear tabla una tabla de traducciones con indentificación "pepe".

2- Agregar las siguientes traducciones una a una:
`000000 =>FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF`
`FFFFFF =>00000000000000000000000000000000`.

3- Agregar las siguientes traducciones como una cadena de bloques
`050505 =>123456789ABCDEF123456789ABCDEF`
`FFFFFF =>112233445566778899AABBCCDDEEFF`
`53FFAA =>111222333444555666777888999AAA`
`10EE05 =>11112222333344445555666677778`8.

4- Borrar las traducciones como bloque `53FFAA` y `FFFFFF`.

5- Calcular las máximas y mínimas claves y valores, e imprimirlos.

6- Imprimir la tabla de traducciones.

7- Imprimir la cantidad de traducciones.

8- Destruir la tabla de traducciones.

Ejercicio 3 - Optativo

Construir un programa que complete de forma aleatoria todas las traducciones posibles de una tabla y calcular el espacio ocupado en memoria. ¿Ocupa lo esperado?.

Testing

En un ataque de bondad, hemos decidido entregarle una serie de *tests* o pruebas para que usted mismo pueda verificar el buen funcionamiento de su código.

Luego de compilar, puede ejecutar `./tester.sh` y eso probará su código. El test realizará una serie de operaciones, estas generarán archivos que serán comparados las soluciones provistas por cátedra. Además, el test realizará pruebas sobre la correcta administración de la memoria dinámica.

Archivos

Se entregan los siguientes archivos:

- `tdt.asm`: Archivo a completar con su código assembler.
- `tdt.c`: Archivo a completar con su código C.
- `main.c`: Archivo para completar la solución del ejercicio 2.
- `Makefile`: Contiene las instrucciones para compilar `tester` y `main`. No debe modificarlo.
- `tdt.h`: Contiene la definición de la estructura tabla de traducciones y adicionales. No debe modificarlo.

- `tester.c`: Codigo de testing. No debe modificarlo.
- `tester.sh`: Script que realiza todos los test. No debe modificarlo.
- `Catedra.salida.caso.chico.txt` y `Catedra.salida.caso.grande.txt`: Resultados de la catedra. No deben modificarlos.

Notas:

a) Para todas las funciones hechas en lenguaje ensamblador que llamen a cualquier función extra, ésta también debe estar hecha en lenguaje ensamblador. (Idem para código C).

b) Toda la memoria dinámica reservada por la función `malloc` debe ser correctamente liberada, utilizando la función `free`.

c) Para el manejo de archivos se recomienda usar las funciones de C: `fopen`, `fread`, `fwrite`, `fclose`, `fseek`, `ftell`, `fprintf`, etc.

d) Para poder correr los test, se debe tener instalado *Valgrind* (En ubuntu: `sudo apt-get install valgrind`).

e) Para corregir los TPs usaremos los mismos tests que les fueron entregados. El criterio de aprobación es que el TP supere correctamente todos los tests y no contenga errores de forma.

3. Informe y forma de entrega

Para este trabajo práctico no deberán entregar un informe. En cambio, deberán entregar un archivo comprimido con el mismo contenido que el dado para realizarlo, habiendo modificado los archivos `tdt.asm` y `tdt.c`, y agregado un archivo `test.c` con el segundo ejercicio resuelto. Además se deberá modificar el *Makefile* para compilar este último.

La fecha de entrega de este trabajo es Martes 12/04. Deberá ser entregado a través de la página web. El sistema sólo aceptará entregas de trabajos hasta las 17:00 hs del día de entrega.

Ante cualquier problema con la entrega, comunicarse por mail a la lista de docentes `orga2-doc@dc.uba.ar`.