Trabajo Práctico 1
Red Caminera

Organización del Computador 2

Primer Cuatrimestre 2017

1. Introducción

Este trabajo práctico consiste en implementar funciones sobre una estructura denominada *RedCaminera*. Una *RedCaminera* es un tipo de datos que almacena ciudades y rutas entre estas ciudades. Ambas entidades, tanto ciudades como rutas serán almacenadas en un contenedor de tipo lista.

Las funciones a implemetar corresponden entonces a una implementación de lista doblemente enlazada y a funciones especificas del tipo *RedCaminera*.

Además de las funciones basicas sobre estos datos, se implementarán funciones mas complejas como combinar redes camineras y generar subredes camineras.

2. Tipo Lista

La estructura de *lista* consiste de dos tipos diferentes de estructuras. Una para almacenar la lista en si y otra para almacenar los nodos. La lista contiene además del puntero al primero y al ultimo nodo de la lista, la longitud de la misma, es decir, la cantidad de nodos.

```
typedef struct lista_t {
  uint32_t longitud;
  struct nodo_t* primero;
  struct nodo_t* ultimo;
} __attribute__((__packed__)) lista;
```

El la estrucutra *nodo* su parte, tiene punteros a los nodos anterior y siguiente y el puntero al dato. Este es un puntero generico, ya que la lista podrá almacenar cualqueir dato. Además el nodo contiene un puntero a la función que se debe ejecutar para borrar el dato en cuestión. Esta función es ejecutada a la hora de borrar toda la lista.

```
typedef struct nodo_t {
  void (*func_borrar)(void*);
  void* dato;
  struct nodo_t* siguiente;
  struct nodo_t* anterior;
} __attribute__((__packed__)) nodo;
```

3. Tipo RedCaminera

La estructura de *RedCaminera* es formada por tres clases distintas de estructuras. La primera, que guarda la información sobre la red caminera en si y dos estructuras mas para almacenar información sobre ciudades y rutas. Tener en cuenta que las listas almacenadas en la estructura de red caminera, corresponden a listas de ciudades y rutas.

```
typedef struct redCaminera_t {
  struct lista_t* ciudades;
  struct lista_t* rutas;
  char* nombre;
} __attribute__((__packed__)) redCaminera;

typedef struct ciudad_t {
  char* nombre;
  uint64_t poblacion;
} __attribute__((__packed__)) ciudad;

typedef struct ruta_t {
  struct ciudad_t* ciudadA;
  double distancia;
  struct ciudad_t* ciudadB;
} __attribute__((__packed__)) ruta;
```

En el contexto de este trabajo, existen una serie de reglas que se deben cumplir para que la red caminera sea valida.

- Una Red Caminera no puede tener dos ciudades con el mismo nombre
- No pueden existir dos ciudades con el mismno nombre
- No pueden existir dos rutas con las mismas ciudades destino
- Las rutas almacenan las ciudades en orden lexicográfico con respecto a sus nombres

En el siguiente gráfico se muestra un ejemplo de las estructuras involucradas en el trabajo.

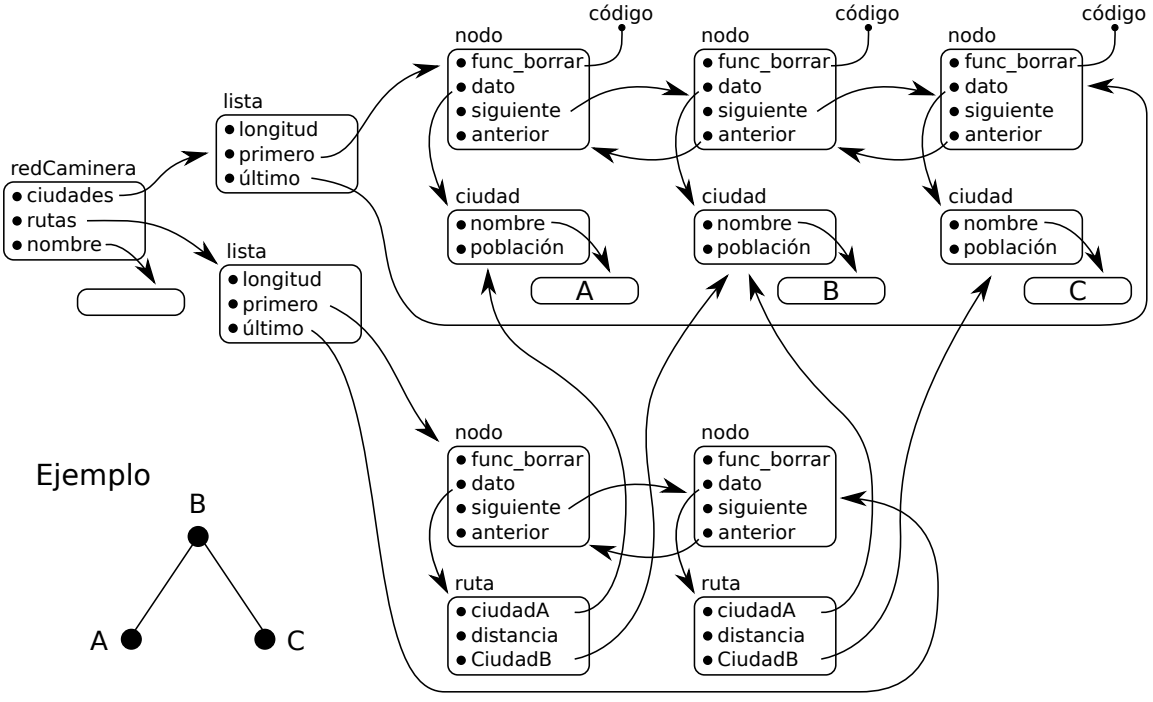

Figura 1: Ejemplo de la estructura de una *RedCaminera*

Funciones de Lista

- `lista* l_crear();`
  Crea una lista vacia.
- `void l_agregarAdelante(lista** l, void* dato, void (*func_borrar)(void*));`
  Agregar un nuevo nodo a la lista como primer elemento.
- `void l_agregarAtras(lista** l, void* dato, void (*func_borrar)(void*));`
  Agregar un nuevo nodo a la lista como último elemento.
- `void l_agregarOrdenado(lista** l, void* dato, void (*func_borrar)(void*), int (*func_cmp)(void*,void*));`
  Agregar un nuevo nodo a la lista de forma ordenada según la función de comparación pasada por parámetro. De existir repetidos, agrega por delante del que resulte igual.
- `void l_borrarTodo(lista* l);`
  Borra la lista, todos sus nodos y datos. Para borrar estos ultimos, en el caso de existir, utiliza la función definida en el nodo. De no tener una función borrar definida en el nodo no se borrará el dato.

Funciones de Red Caminera

Ciudad y Ruta

- `ciudad* c_crear(char* nombre, uint64_t poblacion);`
  Crea una ciudad y completa sus campos. Realiza una copia del nombre.
- `int32_t c_cmp(ciudad* c1, ciudad* c2);`
  Compara dos ciudades por su nombre (ver `str_cmp`)
- `void c_borrar(ciudad* c);`
  Borra una ciudad. Incluyendo el nombre.
- `ruta* r_crear(ciudad* c1, ciudad* c2, double distancia);`
  Crea una ruta y completa sus campos respetando orden lexicográfico entre ciudades, no puede exister una ruta entre la misma ciudad.
- `int32_t r_cmp(ruta* r1, ruta* r2);`
  Compara dos rutas por el nombre de las ciudades. Primero compara por la primer ciudad y en caso de ser iguales compara por la segunda ciudad.
- `void r_borrar(ruta* r);`
  Borra una ruta. No borra las ciudades asociadas.

Red Caminera

- `redCaminera* rc_crear(char* nombre);`
  Crea una red caminera vacia. Copia el nombre de la red.
- `void rc_borrarTodo(redCaminera* rc);`
  Borra toda la red caminera. Tanto ciudades como rutas.
- `void rc_agregarCiudad(redCaminera* rc, char* nombre, uint64_t poblacion);`
  Agrega una nueva ciudad a la red de forma ordenada. En el caso de existir, no agrega nada.

- `void rc_agregarRuta(redCaminera* rc, char* ciudad1, char* ciudad2, double distancia);`
  Agrega una ruta nueva a la red de forma ordenada. En el caso de existir, no agrega nada.

- `void rc_imprimirTodo(redCaminera* rc, FILE *pFile);`
  Imprime toda la red caminera en el descriptor de archivo pasada por parámetro. El formato se describe en la seccición 3.

- `redCaminera* rc_combinarRedes(char* nombre, redCaminera* rc1, redCaminera* rc2);`
  Dadas dos redes camineras, crea una nueva red que combina todas las ciudades y rutas de las mismas. En el caso de existir dos ciudades o rutas iguales, se debe tomar la de la primera red pasada por parámetro. La nueva red caminera llevará el nombre pasado por parámetro.

- `redCaminera* rc_obtenerSubRed(char* nombre, redCaminera* rc, lista* ciudades);`
  Dada una red caminera y una lista de ciudades, crea una nueva red caminera, igual a la pasada por parámetro pero que contiene solamente las ciudades pasadas por parámetro y las rutas entre las mismas. En el caso de no existir una ciudad en la red caminera, pero si en la lista, esta será ignorada. La nueva red caminera llevará el nombre pasado por parámetro.

Consulta sobre Red Caminera

- `ciudad* obtenerCiudad(redCaminera* rc, char* c);`
  Dada una red caminera y un nombre, busca el nombre de ciudad pasado por parámetro y retorna un puntero a la ciudad correspondiente. De no existir la ciudad, retorna *null*.

- `ruta* obtenerRuta(redCaminera* rc, char* c1, char* c2);`
  Dada una red caminera y un par de nombres de ciudades, obtiene el puntero a la ruta que conecta ambas ciudades. De no existir la ruta, retorna *null*.

- `ciudad* ciudadMasPoblada(redCaminera* rc);`
  Retorna el puntero a la ciudad mas poblada de la red caminera pasada por parámetro.

- `ruta* rutaMasLarga(redCaminera* rc);`
  Retorna el puntero a la ruta mas larga de la red caminera pasada por parámetro. De existir mas de una, se toma la de nombre mas chico.

- `void ciudadesMasLejanas(redCaminera* rc, ciudad** c1, ciudad** c2);`
  Modifica los valores de los dobles punteros a ciudad, de forma de cargar en los mismos las ciudades a mas distancia entre si. De existir mas de una, se toma la primer nombre mas chico.

- `uint32_t cantidadDeCaminos(redCaminera *rc, char* ci);`
  Dado el nombre de una ciudad, obtiene la cantidad de rutas que conectan a dicha ciudad.

- `double totalDeDistancia(redCaminera *rc);`
  Obtiene la cantidad total de distancias de todas las rutas de la red.

- `uint64_t totalDePoblacion(redCaminera *rc);`
  Obtiene la cantidad de poblacion de todas las ciudades de la red.

- `ciudad* ciudadMasComunicada(redCaminera *rc);`
  Obtiene el puntero a la ciudad que mas caminos la conectan. De existir mas de una, se toma la de nombre mas chico.

Auxiliares

- `char* str_copy(char* a);`
  Copia una *string* de C terminada en 0.
- `int32_t str_cmp(char* a, char* b);`
  Compara dos *strings* de C terminadas en 0, en orden lexicográfico. Debe retornar:
  - 0 si son iguales
  - 1 si `a` $<$ `b`
  - -1 si `b` $<$ `a`

Detalle sobre funciones

La función `rc_imprimirTodo` debe imprimir la red caminera respetando el siguiente formato:

`Nombre:` *nombre de la red caminera*
`Ciudades: [` *nombre de ciudad 1* `,` *poblacion de ciudad 1* `] ... [ ... , ... ]`
`Rutas: [` *puntero a ciudad A* `,` *puntero a ciudad B* `,` *distancia entre ciudad A y B* `] ... [ ... , ... , ... ]`

Ejemplo:

```
Nombre:
Europa
Ciudades:
[Barcelona,1608746]
[Berlin,3469849]
[Roma,2874038]
[Paris,2240621]
Rutas:
[Barcelona,Berlin,1861.1]
[Barcelona,Roma,1354.2]
[Barcelona,Paris,1035.7]
[Berlin,Roma,1500.6]
[Berlin,Paris,1054.7]
[Roma,Paris,1421.5]
```

Todos los numeros en punto flotante debe imprimirse con un solo decimal.

4. Enunciado

Ejercicio 1

Implementar todas las funciones mencionadas anteriormente según se indica a continuación:

En lenguaje assembler

- `lista* l_crear();` *(8 líneas)*
- `void l_agregarAdelante(lista** l, void* dato, void (*func_borrar)(void*));` *(29 líneas)*
- `void l_agregarAtras(lista** l, void* dato, void (*func_borrar)(void*));` *(29 líneas)*

- `void l_agregarOrdenado(lista** l, void* dato, void (*func_borrar)(void*), int (*func_cmp)(void*,void*));` *(61 líneas)*
- `void l_borrarTodo(lista* l);` *(22 líneas)*
- `ciudad* c_crear(char* nombre, uint64_t poblacion);` *(17 líneas)*
- `int32_t c_cmp(ciudad* c1, ciudad* c2);` *(3 líneas)*
- `void c_borrar(ciudad* c);` *(7 líneas)*
- `ruta* r_crear(ciudad* c1, ciudad* c2, double distancia);` *(30 líneas)*
- `int32_t r_cmp(ruta* r1, ruta* r2);` *(15 líneas)*
- `void r_borrar(ruta* r);` *(1 líneas)*
- `redCaminera* rc_crear(char* nombre);` *(21 líneas)*
- `void rc_borrarTodo(redCaminera* rc);` *(11 líneas)*
- `void rc_agregarCiudad(redCaminera* rc, char* nombre, uint64_t poblacion);` *(24 líneas)*
- `void rc_agregarRuta(redCaminera* rc, char* ciudad1, char* ciudad2, double distancia);` *(37 líneas)*
- `ciudad* obtenerCiudad(redCaminera* rc, char* c);` *(17 líneas)*
- `ruta* obtenerRuta(redCaminera* rc, char* c1, char* c2);` *(31 líneas)*
- `ciudad* ciudadMasPoblada(redCaminera* rc);` *(11 líneas)*
- `ruta* rutaMasLarga(redCaminera* rc);` *(11 líneas)*
- `void ciudadesMasLejanas(redCaminera* rc, ciudad** c1, ciudad** c2);` *(7 líneas)*
- `uint32_t cantidadDeCaminos(redCaminera *rc, char* ci);` *(17 líneas)*
- `double totalDeDistancia(redCaminera *rc);` *(10 líneas)*
- `uint64_t totalDePoblacion(redCaminera *rc);` *(10 líneas)*
- `ciudad* ciudadMasComunicada(redCaminera *rc);` *(31 líneas)*
- `char* str_copy(char* a);` *(27 líneas)*
- `int32_t str_cmp(char* a, char* b);` *(23 líneas)*

En lenguaje C

- `void rc_imprimirTodo(redCaminera* rc, FILE *pFile);` *(14 líneas)*
- `redCaminera* rc_combinarRedes(char* nombre, redCaminera* rc1, redCaminera* rc2);` *(23 líneas)*
- `redCaminera* rc_obtenerSubRed(char* nombre, redCaminera* rc, lista* ciudades);` *(14 líneas)*

Entre paréntesis se menciona de forma ilustrativa la cantidad de líneas 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 una red caminera de nombre "kukamonga"

2- Agregar las ciudades [1] "montebello"(12041 habitantes), [2]"north haverbrook"(1244 habitantes) y [3] "cocula"(342 habitantes)

3- Agregar las siguientes rutas: [1] − [2] = 232km, [1] − [3] = 233km, [2] − [3] = 236km

4- Obtener "ciudadMasPoblada" y "rutaMasLarga"

5- Imprimir el resultado en el archivo "PepeGuapo.txt"

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:

- `redcaminera.asm`: Archivo a completar con su código assembler.
- `redcaminera.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.
- `redcaminera.h`: Contiene la definición de la estructura y funciones. 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.`X`.txt` : Resultados de la cátedra para cada caso X. 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.

5. 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 `cuatrotree.asm`, `cuatrotree.c` y `main.c`.

La fecha de entrega de este trabajo es 18/04. Deberá ser entregado a través de un repositorio GIT almacenado en `https://git.exactas.uba.ar` respetando el protocolo enviado por mail y publicado en la página web de la materia.
Ante cualquier problema con la entrega, comunicarse por mail a la lista de docentes `orga2-doc@dc.uba.ar`.