Trabajo Práctico 1
Sorteador

Organización del Computador II

Primer Cuatrimestre de 2020

1. Introducción

El objetivo de este TP es implementar un conjunto de funciones sobre distintas estructuras de datos. Consideraremos dos estructuras básicas, denominadas `String` y `List`, y una estructura más compleja denominada `Sorter`.

Las funciones de `String` serán operaciones sobre cadenas de caracteres terminadas en cero. Las de `Lists` permitirán crear, agregar, borrar e imprimir elementos de una lista doblemente enlazada.

La estructura `Sorter` en cambio, será básicamente un vector de listas que permita ordernar los datos en las diferentes listas dependiendo de una función que determine en cual lista guardar el dato.

A continuación se explicará en detalle el funcionamiento de cada una de estas estructuras y sus funciones asociadas.

2. Estructuras

Previo a describir las estructuras de datos, es necesario conocer los tipos de las funciones que serán pasadas como parametros. Algunas de las funciones a implementar requieren como parámetro punteros a otras funciones. Un ejemplo de esto es la función imprimir de listas, que toma una función que permite imprimir cada dato de la lista.
Las aridades de los punteros a función pasados por parámetro son las siguientes:

- Función borrar: `typedef void (funcDelete_t)(void*);`
- Función imprimir: `typedef void (funcPrint_t)(void*, FILE *pFile);`
- Función comparar: `typedef int32_t (funcCmp_t)(void*, void*);`
- Función duplicar: `typedef void* (funcDup_t)(void*);`
- Función ordenar: `typedef int32_t (funcSorter_t)(void*);`

Estructura `String`

Este tipo no es una estructura en sí misma sino un puntero a un string de C terminado en cero.

Las funciones para operar con este son:

- `char* strClone(char* a)`
  Genera una copia del *string* pasado por parámetro. El puntero pasado siempre es válido aunque podría corresponderse a la *string* vacía.
- `uint32_t strLen(char* a)`
  Retorna la cantidad de caracteres distintos de cero del *string* pasado por parámetro.
- `int32_t strCmp(char* a, char* b)`
  Compara dos *strings* en orden lexicográfico[1]. Debe retornar:

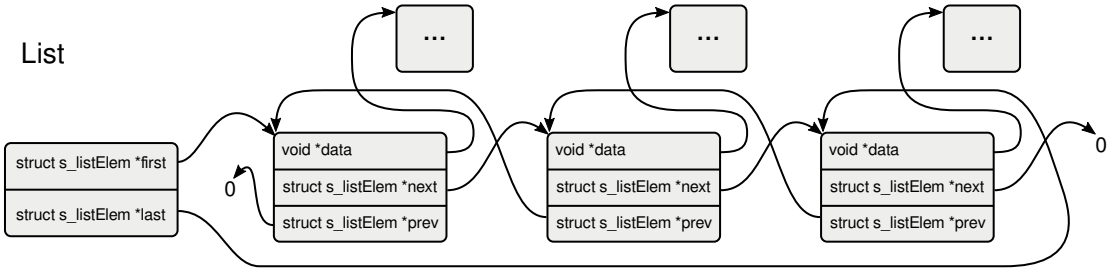
[1] `https://es.wikipedia.org/wiki/Orden_lexicografico`

- 0 si son iguales
- 1 si `a` < `b`
- −1 si `b` < `a`

- `char* strConcat(char* a, char* b)`
  Genera un nuevo *string* con la concatenación de `a` y `b`, libera la memoria ocupada por estos últimos.
- `void strDelete(char* a)`
  Borra el *string* pasado por parámetro. Esta función es equivalente a la función `free`.
- `void strPrint(char* a, FILE *pFile)`
  Escribe el *string* en el *stream* indicado a través de `pFile`. Si el *string* es vacío debe escribir "`NULL`".
- `void hexPrint(char* a, FILE *pFile)`
  Escribe el *string* en el *stream* indicado a través de `pFile` en formato hexadecimal.

Estructura `List`

Implementa una lista doblemente enlazada de nodos. La estructura `lista_t` contiene un puntero al primer y último elemento de la lista, mientras que cada elemento es un nodo de tipo `listElem_t` que contiene un puntero a su siguiente, anterior y al dato almacenado. Este último es de tipo `void*`, permitiendo referenciar cualquier tipo de datos.




```
typedef struct s_listElem{
    void *data;
    struct s_listElem *next;
    struct s_listElem *prev;
} listElem_t;

typedef struct s_list{
    struct s_listElem *first;
    struct s_listElem *last;
} list_t;
```

- `list_t* listNew()`
  Crea una nueva `list_t` vacía donde los punteros a `first` y `last` estén inicializados en cero.
- `void listAddFirst(list_t* l, void* data)`
  Agrega un nuevo nodo al principio de la lista, que almacene `data`.
- `void listAddLast(list_t* l, void* data)`
  Agrega un nuevo nodo al final de la lista, que almacene `data`.
- `void listAdd(list_t* l, void* data, funcCmp_t* fc)`
  Agrega un nuevo nodo que almacene `data`, respetando el orden dado por la función `f`.
- `void listRemoveFirst(list_t* l, funcDelete_t* fd)`
  Borra el primer nodo de la lista. Si `fd` no es cero, utiliza la función para borrar el dato correspondientemente.
- `void listRemoveLast(list_t* l, funcDelete_t* fd)`
  Borra el último nodo de la lista. Si `fd` no es cero, utiliza la función para borrar el dato correspondientemente.

- `void listRemove(list_t* l, void* data, funcCmp_t* fc, funcDelete_t* fd)`
  Borra todos los nodos de la lista cuyo dato sea igual al contenido de `data` según la función de comparación apuntada por `fc`. Si `fd` no es cero, utiliza la función para borrar los datos en cuestión.

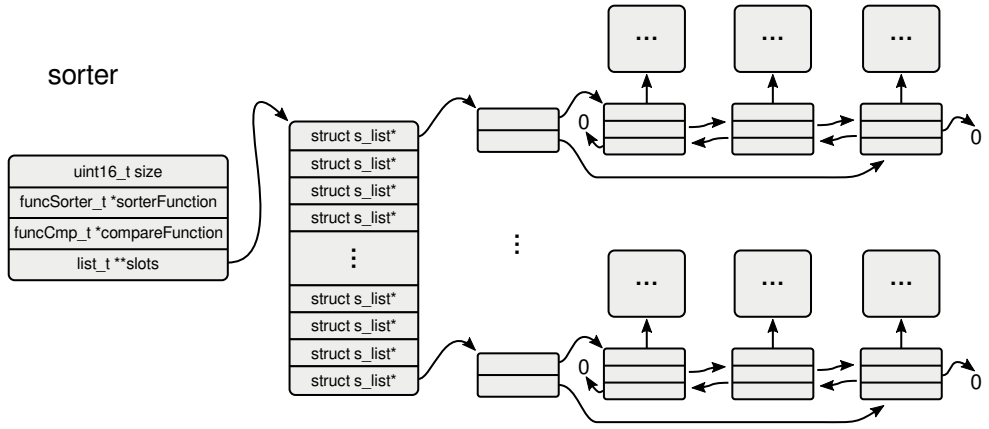
- `list_t* listClone(list_t* l, funcDup_t* fn)`
  Construye una copia de la lista, copiando todos los elementos de la misma mediante la función pasada por parámetro.

- `void listDelete(list_t* l, funcDelete_t* fd)`
  Borra la lista completa con todos sus nodos. Si `fd` no es cero, utiliza la función para borrar sus datos correspondientemente.

- `void listPrint(list_t* l, FILE *pFile, funcPrint_t* fp)`
  Escribe en el *stream* indicado por `pFile` la lista almacenada en `l`. Para cada dato llama a la función `fp`, y si esta es cero, escribe el puntero al dato con el formato `"%p"`. El formato de la lista será: `[`x_0`,...,`x_{n-1}`]`, suponiendo que x_i es el resultado de escribir el i-ésimo elemento.

Estructura `sorter`

La estructura `sorter_t` almacena un arreglo de listas de tamaño `size`, a partir del puntero `slots`. Sobre cada una de las listas de este arreglo, almacenará cada uno de los datos cargados mediante la función `sorterAdd`. Para cargar un dato dentro de una estructura `sorter_t`, se llamará a la función `sorterFunction` pasando como parámetro el dato en cuestión. La función retornará un número que indicará el `slot` (indice en el arreglo `slots`) donde debe ser agregado el dato. La estructura de datos soportará datos duplicados.




```
typedef struct s_sorter{
    uint16_t size;
    funcSorter_t *sorterFunction;
    funcCmp_t *compareFunction;
    list_t **slots;
} sorter_t;
```

- `sorter_t* sorterNew(uint16_t slots, funcSorter_t* fs, funcCmp_t* fc)`
  Crea una nueva `sorter_t` vacía. Los punteros `funcSorter_t` y `funcCmp_t` son cargados por parámetro. El tamaño del arreglo `slots` es cargado en la estructura como `size`. Es recomendable inicializar el arreglo `slots` con listas vacias, aunque se podría indicar al momento de cargar el primer elemento en el slot.

- `void sorterAdd(sorter_t* sorter, void* data)`
  Usando la función `sorterFunction` determina el slot donde agregar el elemento `data`. Una vez seleccionado, agrega de forma ordenada en al lista apuntada por el correspondiente slot. Si el elemento ya existe, entonces lo agrega nuevamente.

- `void sorterRemove(sorter_t* sorter, void* data, funcDelete_t* fd)`
  Usando la función `sorterFunction` determina el slot de donde borrar el elemento `data`. Una vez seleccionado, borra la primer aparición del elemnto en la lista apuntada por el correspondiente slot.

- `list_t* sorterGetSlot(sorter_t* sorter, uint16_t slot, funcDup_t* fn)`
  Retorna una copia de de la lista almacenada en el indice `slot`. Utilizando la función `fn`, duplica uno a uno los elementos almacenados en la lista.

- `char* sorterGetConcatSlot(sorter_t* sorter, uint16_t slot)`
  Dado un `slot`, obtiene una string que concatena todos los elementos de la lista de forma ordenada.

- `void sorterCleanSlot(sorter_t* sorter, uint16_t slot, funcDelete_t* fd)`
  Borra todos los elementos almacenados en `slot`. Utiliza la función `fd` para borrar los datos dentro de la lista.

- `void sorterDelete(sorter_t* sorter, funcDelete_t* fd)`
  Borra todos los datos de una estructura `sorter_t`.

- `void sorterPrint(sorter_t* sorter, FILE *pFile, funcPrint_t* fp)`
  Escribe en el *stream* indicado por `pFile` el arreglo de listas almacenado en `sorter`. Las listas serán escritas en líneas distintas respetando el siguiente formato: $i = [x_0,...,x_{n-1}]$, donde x_i es el resultado de escribir el i-ésimo dato. Para cada dato llama a la función `fp`.

Funciones `sorter`

- `uint16_t fs_sizeModFive(char* s)`
  Obtiene la longitud de `s` y aplica el modulo en base 5.

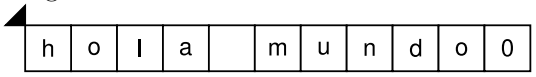
- `uint16_t fs_firstChar(char* s)`
  Obtiene el valor del primer caracter de la `s`. Si es nula, entonces el valor es cero.

- `uint16_t fs_bitSplit(char* s)`
  Obtiene un número entre 0 y 9. Si el primer caracter es cero, entonces retorna 8. Si el valor es una potencia de 2, entonces retorna el indice del bit en 1 para su representación binaria. En cualquier otro caso retorna 9.
  Ejmplos:

```
fs_bitSplit([00:...]) -> 8
fs_bitSplit([01:...]) -> 0
fs_bitSplit([02:...]) -> 1
fs_bitSplit([04:...]) -> 2
fs_bitSplit([08:...]) -> 3
fs_bitSplit([10:...]) -> 4
fs_bitSplit([20:...]) -> 5
fs_bitSplit([40:...]) -> 6
fs_bitSplit([80:...]) -> 7
fs_bitSplit([A3:...]) -> 9
```

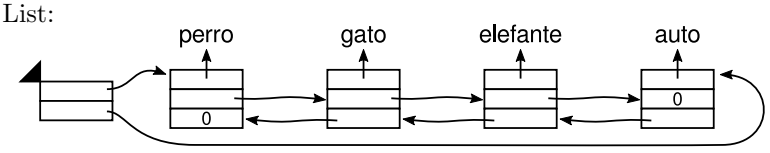
Ejemplos

- String:



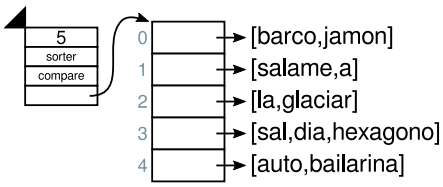
  Print: `hola mundo`

- List:



  Print: `[perro,gato,elefante,auto]`

- Sorter:



Print:

```
0 = [barco,jamon]
1 = [salame,a]
2 = [la,glaciar]
3 = [sal,dia,hexagono]
4 = [auto,bailarina]
```

3. Enunciado

A continuación se detallan las funciones a implementar en lenguaje ensamblador.

- `char* strClone(char* a)`(19 Inst.)
- `uint32_t strLen(char* a)`(7 Inst.)
- `int32_t strCmp(char* a, char* b)`(25 Inst.)
- `char* strConcat(char* a, char* b)`(42 Inst.)
- `void strDelete(char* a)`(1 Inst.)
- `void strPrint(char* a, FILE *pFile)`(7 Inst.)
- `list_t* listNew()`(7 Inst.)
- `void listAddFirst(list_t* l, void* data)`(21 Inst.)
- `void listAddLast(list_t* l, void* data)`(21 Inst.)
- `void listAdd(list_t* l, void* data, funcCmp_t* fc)`(54 Inst.)
- `list_t* listClone(list_t* l, funcDup_t* fn)`(31 Inst.)
- `void listDelete(list_t* l, funcDelete_t* fd)`(26 Inst.)
- `void listPrint(list_t* l, FILE *pFile, funcPrint_t* fp)`(40 Inst.)
- `sorter_t* sorterNew(uint16_t slots, funcSorter_t* fs, funcCmp_t* fc)`(36 Inst.)
- `void sorterAdd(sorter_t* sorter, void* data)`(14 Inst.)
- `void sorterRemove(sorter_t* sorter, void* data, funcDelete_t* fd)`(8 Inst.)
- `list_t* sorterGetSlot(sorter_t* sorter, uint16_t slot, funcDup_t* fn)`(5 Inst.)
- `char* sorterGetConcatSlot(sorter_t* sorter, uint16_t slot)`(25 Inst.)
- `void sorterCleanSlot(sorter_t* sorter, uint16_t slot, funcDelete_t* fd)`(18 Inst.)
- `void sorterDelete(sorter_t* sorter, funcDelete_t* fd)`(21 Inst.)
- `void sorterPrint(sorter_t* sorter, FILE *pFile, funcPrint_t* fp)`(36 Inst.)
- `uint16_t fs_sizeModFive(char* s)`(11 Inst.)
- `uint16_t fs_firstChar(char* s)`(2 Inst.)
- `uint16_t fs_bitSplit(char* s)`(15 Inst.)

Además las siguientes funciones deben ser implementadas en lenguaje C.

- `void listRemoveFirst(list_t* l, funcDelete_t* fd)`(6 Líneas)
- `void listRemoveLast(list_t* l, funcDelete_t* fd)`(6 Líneas)
- `void listRemove(list_t* l, void* data, funcCmp_t* fc, funcDelete_t* fd)`(17 Líneas)

Por último, la siguiente función viene dada por la cátedra.

- `void hexPrint(char* a, FILE *pFile)`

Recordar que cualquier función auxiliar que desee implementar debe estar en lenguaje ensamblador. A modo de referencia, se indica entre paréntesis la cantidad de instrucciones necesarias para resolver cada una de las funciones.

Compilación y Testeo

Para compilar el código y poder correr las pruebas cortas deberá ejecutar `make main` y luego `./runMain.sh`. En cambio, para compilar el código y correr las pruebas intensivas deberá ejecutar `./runTester.sh`.

Pruebas cortas

Deberá construirse un programa de prueba modificando el archivo `main.c` para que realice las acciones detalladas a continuación, una después de la otra:

1- **Caso list**

- Crear una lista vacío.
- Agregar exactamente 10 strings cualquiera.
- Imprimir la lista.
- Borrar la lista.

2- **Caso sorter**

- Crear un sorter vacío utilizando la función `fs_sizeModFive(char* s)`.
- Agregar un string en todos los slots.
- Imprimir el sorter.

El programa puede correrse con `./runMain.sh` para verificar que no se pierde memoria ni se realizan accesos incorrectos a la misma.

Pruebas intensivas (Testing)

Entregamos también una serie de *tests* o pruebas intensivas para que pueda verificarse el buen funcionamiento del código de manera automática. Para correr el testing se debe ejecutar `./runTester.sh`, que compilará el *tester* y correrá todos los tests de la cátedra. Un test consiste en la creación, inserción, eliminación, ejecución de funciones e impresión en archivos de alguna estructura implementada. Luego de cada test, el *script* comparará los archivos generados por su TP con las soluciones correctas provistas por la cátedra. También será probada la correcta administración de la memoria dinámica.

Notas

- Toda la memoria dinámica reservada usando la función `malloc` debe ser correctamente liberada, utilizando la función `free`.
- Para el manejo de archivos se recomienda usar las funciones de `C`: `fopen`, `fprintf`, `fwrite`, `fputc`, `fputs` y `fclose`.
- Para el manejo de strings **no está permitido** usar las funciones de `C`: `strlen`, `strcpy`, `strcmp` y `strdup`.
- Para correr los tests deben tener instalado *Valgrind* (en Ubuntu: `sudo apt-get install valgrind`).
- Para corregir los TPs usaremos los mismos tests que les fueron entregados. Es condición necesaria para la aprobación que el TP supere correctamente todos los tests.

Archivos

Se entregan los siguientes archivos:

- `lib.h`: Contiene la definición de las estructuras y de las funciones a realizar. No debe ser modificado.
- `lib.asm`: Archivo a completar con la implementación de las funciones en lenguaje ensamblador.
- `lib.c`: Archivo a completar con la implementación de las funciones en lenguaje C.

- `Makefile`: Contiene las instrucciones para ensamblar y compilar el código.
- `main.c`: Archivo donde escribir los ejercicios para las pruebas cortas (`runMain.sh`).
- `tester.c`: Archivo del tester de la cátedra. No debe ser modificado.
- `runMain.sh`: Script para ejecutar el test simple (pruebas cortas). No debe ser modificado.
- `runTester.sh`: Script para ejecutar todos los tests intensivos. No debe ser modificado.
- `salida.caso*.catedra.txt`: Archivos de salida para comparar con sus salidas. No debe ser modificado.

4. Informe y forma de entrega

Para este trabajo práctico no deberán entregar un informe. En cambio, deberán entregar el mismo contenido que fue dado para realizarlo, habiendo modificado **solamente** los archivos `lib.asm`, `lib.c` y `main.c`.

La fecha de entrega de este trabajo es Martes 12/05. 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 (`https://campus.exactas.uba.ar/`). A la hora de subir su trabajo al GIT es fuertemente recomendable que ejecuten la operación `make clean` borrando todos los binarios e impidiendo que estos sean cargados al repositorio.
Ante cualquier problema con la entrega, comunicarse por mail a la lista de docentes `orga2-doc@dc.uba.ar`.