

# Trabajo Práctico 1

## Contenedores locos

Organización del Computador II

Segundo Cuatrimestre de 2019

### 1. Introducción

El objetivo de este TP es implementar un conjunto de funciones sobre distintas estructuras de datos. Consideraremos dos estructuras básicas y una estructura compleja, denominadas **String**, **List** y **HashTable** respectivamente.

Las funciones de **String** serán operaciones sobre cadenas de caracteres, cuya finalización está determinada por la aparición del *caracter nulo*, es decir, las cadenas terminan en cero. Las funciones de **Lists** permitirán crear, agregar, borrar e imprimir elementos de una lista doblemente enlazada. La estructura **HashTable** será un vector de listas que permita guardar datos en listas indexadas sobre un vector de tamaño fijo.

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

### 2. Estructuras

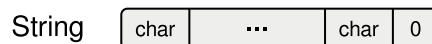
Las estructuras **List** y **HashTable** pueden contener cualquier tipo de datos. Para operar con estos elementos es necesario conocer a que función llamar en cada caso. Para resolver esto, las funciones toman como parámetros punteros a otras funciones. Un ejemplo de esto es la función imprimir de listas, 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 de hash: `typedef int32_t (funcHash_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* pString)`  
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* pString)`  
Retorna la cantidad de caracteres, contando desde el primer caracter hasta el último sin el caracter nulo (cero).
- `int32_t strCmp(char* pStringA, char* pStringB)`  
Compara dos *strings* en orden lexicográfico<sup>1</sup>. Debe retornar:

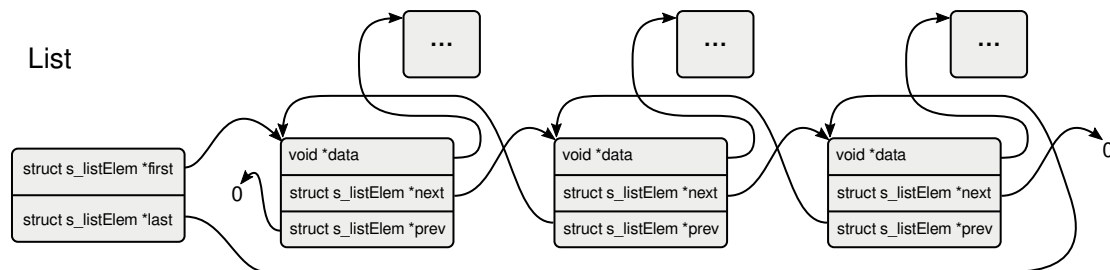
---

<sup>1</sup>[https://es.wikipedia.org/wiki/Orden\\_lexicografico](https://es.wikipedia.org/wiki/Orden_lexicografico)

- 0 si son iguales
  - 1 si  $a < b$
  - -1 si  $b < a$
- `char* strConcat(char* pStringA, char* pStringB)`  
 Genera un nuevo *string* con la concatenación de *a* y *b*, libera la memoria ocupada por estos últimos.
  - `char* strSubstring(char* pString, uint32_t inicio, uint32_t fin)`  
 Genera un nuevo *string* tomando los caracteres desde el índice *inicio* hasta el índice *fin* inclusive, liberando la memoria ocupada por la *string* pasada por parámetro. Si  $\text{inicio} > \text{fin}$ , retorna el mismo *string* pasado por parámetro. Considerando *len* como la cantidad de caracteres del *string*, si  $\text{inicio} > \text{len}$ , entonces retorna la *string* vacía. Si  $\text{fin} > \text{len}$ , se tomará como límite superior la longitud del *string*. Ejemplos: `strSubstring("ABC", 1, 1) → "B"`,  
`strSubstring("ABC", 10, 0) → "ABC"`,  
`strSubstring("ABC", 2, 10) → "C"`
  - `void strDelete(char* pString)`  
 Borra el *string* pasado por parámetro. Esta función es equivalente a la función `free`.
  - `void strPrint(char* pString, FILE *pFile)`  
 Escribe el *string* en el *stream* indicado a través de *pFile*. Si el *string* es vacío debe escribir "NULL".

## Estructura List

Implementa una lista doblemente enlazada de nodos. La estructura `list_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;
```

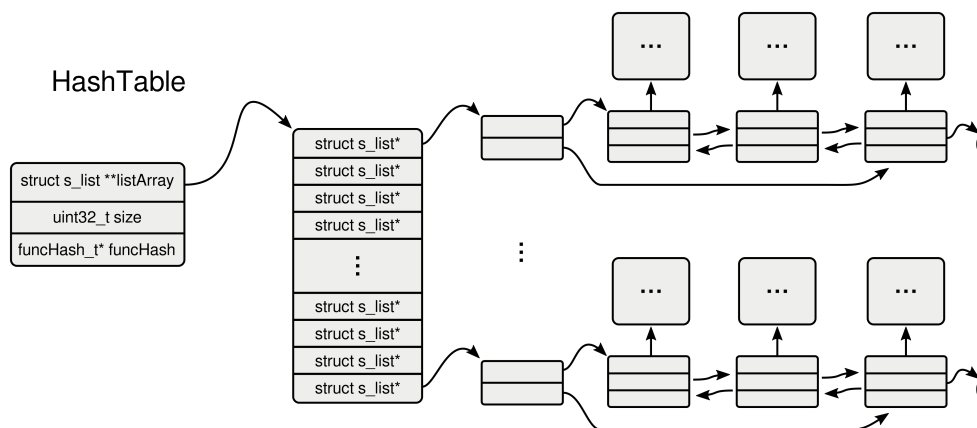
- `list_t* listNew()`  
 Crea una nueva `list_t` vacía donde los punteros a `first` y `last` estén inicializados en cero.
- `void listAddFirst(list_t* pList, void* data)`  
 Agrega al principio de la lista un nuevo nodo que almacena *data*.
- `void listAddLast(list_t* pList, void* data)`  
 Agrega un nuevo nodo al final de la lista que almacena *data*.
- `void listAdd(list_t* pList, void* data, funcCmp_t* fc)`  
 Agrega un nuevo nodo que almacene *data*, respetando el orden dado por la función *fc*. Para este ítem suponer que la lista viene ordenada crecientemente, debiéndose mantener esta condición luego de la inserción de *data*.

- `void listRemoveFirst(list_t* pList, 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* pList, 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* pList, 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.
- `void listDelete(list_t* pList, 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* pList, FILE *pFile, funcPrint_t* fp)`  
Escribe en el *stream* indicado por `pFile` la lista almacenada en `pList`. 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:  $[x_0, \dots, x_{n-1}]$ , suponiendo que  $x_i$  es el resultado de escribir el *i*-ésimo elemento.
- `void listPrintReverse(list_t* pList, FILE *pFile, funcPrint_t* fp)`  
Realiza la misma tarea que la función `listPrint` pero escribiendo los elementos de la lista en orden inverso.

## Estructura HashTable

La estructura `hashTable_t` almacena un arreglo de listas de tamaño `size`, a partir del puntero `listArray`. Llamaremos a cada lista `slot`. Cada vez que guardemos un dato en la `HashTable`, éste irá a parar a un `slot` específico. El `slot` que le corresponde a cada dato estará dado por una función de hash que definiremos al momento de crear la estructura. Si el número de `slot` que decide la función de hash para un dato es mayor que el tamaño del arreglo, se tomará el módulo del tamaño total del arreglo (es decir, el resto de dividir por este tamaño).

Un ejemplo clásico de función de hash para *strings* es mapear la primera letra de la palabra a un número que corresponda con su posición en el alfabeto. Es decir, `hash("abeja") → 1`, `hash("arbol") → 1`, `hash("casa") → 3`



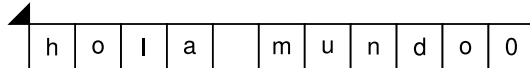
```
typedef struct s_hashTable{
    struct s_list **listArray;
    uint32_t size;
    funcHash_t* funcHash;
} hashTable_t;
```

- `hashTable_t* hashTableNew(uint32_t size, funcHash_t* funcHash)`  
Crea un nuevo `hashTable_t` con una función de hash `funcHash` y un arreglo de tamaño `size`, completándolo con listas vacías.

- `void hashTableAdd(hashTable_t* pTable, void* data)`  
Agrega un nuevo elemento que contenga `data` en el `slot` determinado por la función de hash (módulo el tamaño del arreglo). El orden interno de los elementos en cada `slot` no está determinado.
- `void hashTableRemoveAll(hashTable_t* pTable, void* data, funcCmp_t* fc, funcDelete_t* fd)`  
Borra todas las apariciones de `data` del `slot` al que pertenece. Si el valor de `fd` no es cero, utiliza la función para borrar los datos dados.
- `void hashTableDeleteSlot(hashTable_t* pTable, uint32_t slot, funcDelete_t* fd)`  
Borra todos los elementos del `slot` indicado. Si el valor de `fd` no es cero, utiliza la función para borrar los datos dados.
- `void hashTableDelete(hashTable_t* pTable, funcDelete_t* fd)`  
Borra todas las estructuras apuntadas desde `hashTable_t`. Si `fd` no es cero, utiliza la función para borrar sus datos respectivamente.
- `void hashTablePrint(hashTable_t* pTable, FILE *pFile, funcPrint_t* fp)`  
Escribe en el *stream* indicado por `pFile` la tabla de listas almacenada en `t`. Para cada dato llama a la función `fp`, y si es cero, escribe el puntero al dato con el formato "%p". Cada elemento del arreglo contiene una lista. Éstas serán escritas en líneas distintas respetando el siguiente formato:  $i = [x_0, \dots, x_{n-1}]$ , donde  $x_i$  es el resultado de escribir el  $i$ -ésimo dato.

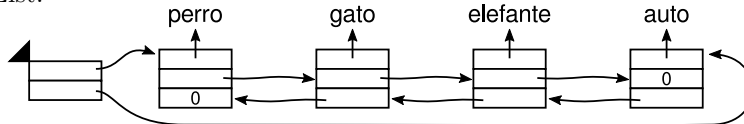
## Ejemplos

- String:



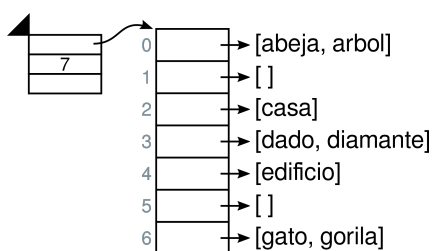
Print: hola mundo

- List:



Print: [perro,gato,elefante,auto]

- HashTable:



Print:

```
0 = [abeja, arbol]
1 = []
2 = [casa]
3 = [dado, diamante]
4 = [edificio]
5 = []
6 = [gato, gorila]
```

## 3. Enunciado

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

- `char* strClone(char* pString)`(19 Inst.)
- `uint32_t strLen(char* pString)`(7 Inst.)
- `int32_t strCmp(char* pStringA, char* pStringB)`(25 Inst.)
- `char* strConcat(char* pStringA, char* pStringB)`(42 Inst.)
- `void strDelete(char* pString)`(1 Inst.)
- `void strPrint(char* pString, FILE *pFile)`(7 Inst.)

- `list_t* listNew()`(7 Inst.)
- `void listAddFirst(list_t* pList, void* data)`(21 Inst.)
- `void listAddLast(list_t* pList, void* data)`(21 Inst.)
- `void listAdd(list_t* pList, void* data, funcCmp_t* fc)`(54 Inst.)
- `void listRemoveFirst(list_t* pList, funcDelete_t* fd)`(16 Inst.)
- `void listRemoveLast(list_t* pList, funcDelete_t* fd)`(16 Inst.)
- `void listRemove(list_t* pList, void* data, funcCmp_t* fc, funcDelete_t* fd)`(49 Inst.)
- `void listDelete(list_t* pList, funcDelete_t* fd)`(26 Inst.)
- `void listPrint(list_t* pList, FILE *pFile, funcPrint_t* fp)`(40 Inst.)
- `hashTable_t* hashTableNew(uint32_t size, funcHash_t* funcHash)`(32 Inst.)
- `void hashTableAdd(hashTable_t* pTable, void* data)`(13 Inst.)
- `void hashTableDeleteSlot(hashTable_t* pTable, uint32_t slot, funcDelete_t* fd)`(21 Inst.)
- `void hashTableDelete(hashTable_t* pTable, funcDelete_t* fd)`(21 Inst.)

Además las siguientes funciones deben ser implementadas en lenguaje C.

- `char* strSubstring(char* pString, uint32_t inicio, uint32_t fin)`(15 líneas)
- `void listPrintReverse(list_t* pList, FILE *pFile, funcPrint_t* fp)`(15 líneas)
- `void hashTableRemoveAll(hashTable_t* pTable, void* data, funcCmp_t* fc, funcDelete_t* fd)`(2 líneas)
- `void hashTablePrint(hashTable_t* pTable, FILE *pFile, funcPrint_t* fp)`(5 líneas)

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:

- Crear una HashTable vacía de tamaño 33.
- Agregar un string en todos los slots.
- Imprimir la HashTable.

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 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 05/09. 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/course/view.php?id=1464>). 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](mailto:orga2-doc@dc.uba.ar).