# Trabajo Práctico 1
# Contenedores locos

Organización del Computador II

Primer 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, denominadas `String` y `List`, y dos estructuras complejas, `n3tree` y `nTable`.

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 `n3tree` consistirá en un árbol casi binario, con la particularidad de guardar elementos iguales en una lista, mientras que la estructura `nTable` 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.

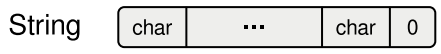
## 2. Estructuras

Las estructuras `List`, `n3tree` y `nTable` 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*);`

### 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:
  - 0 si son iguales

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

- 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.

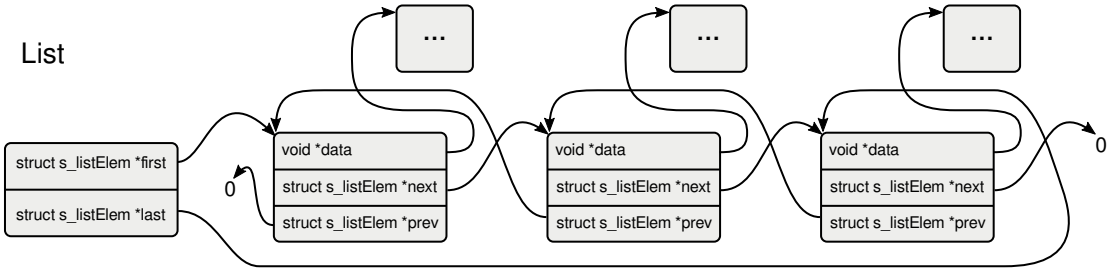
- `char* strRange(char* a, uint32_t i, uint32_t f)`
  Genera un nuevo *string* tomando los caracteres del índice `i` al `f` inclusive, libera la memoria ocupada por la *string* pasada por parámetro. Si `i`>`f`, retorna el mismo *string* pasado por parámetro. Considerando *len* como la cantidad de caracteres del *string*, si `i`>*len*, entonces retorna la *string* vacía. Si `f`>*len*, se tomará como límite superior la longitud del *string*. Ejemplos: `strRange("ABC", 1, 1)` → `"B"`, `strRange("ABC", 10, 0)` → `"ABC"`, `strRange("ABC", 2, 10)` → `"C"`

- `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`".

## 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.

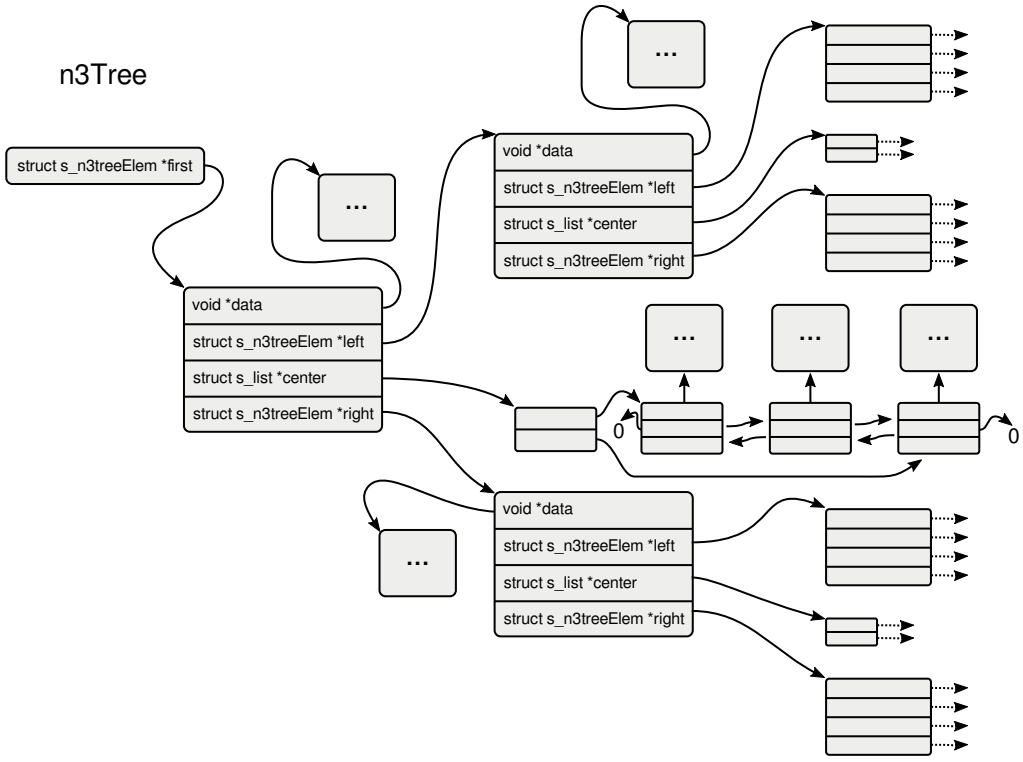
- `void listDelete(list_t* l, funcDelete_t* fd)`
  Borra la lista completa con todos sus nodos. Si `fd` no es cero, utiliza la función para borrar sus datos correspondientemente.

- `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.

- `void listPrintReverse(list_t* l, FILE *pFile, funcPrint_t* fp)`
  Realiza la misma tarea que la función `listPrint` pero escribiendo los elementos de la lista en orden inverso.

## Estructura `n3tree`

Esta estructura representa un árbol binario de busqueda. El tipo `n3tree_t` almacena el puntero al primer elemento del árbol (su raíz), mientras que `n3treeElem_t` almacena cada uno de los nodos internos. La estructura `n3treeElem_t` contiene un puntero al dato, uno a su subárbol derecho, otro al izquierdo y uno a una lista denominado `center`. Tanto el subárbol derecho como el izquierdo deben preservar el invariante de que todo dato del subarbol derecho es mayor al dato almacenado en el nodo, y todo dato del subarbol izquierdo es menor al dato almacenado en el nodo. Por su parte, la lista `center` debe almacenar todos los datos que sean iguales al almacenado en el nodo.



```
typedef struct s_n3tree{
    struct s_n3treeElem *first;
} n3tree_t;

typedef struct s_n3treeElem{
    void* data;
    struct s_n3treeElem *left;
    struct s_list *center;
    struct s_n3treeElem *right;
} n3treeElem_t;
```

- `n3tree_t* n3treeNew()`
  Crea un `n3tree_t` vacío.

- `void n3treeAdd(n3tree_t* t, void* data, funcCmp_t* fc)`
  Crea un nuevo nodo donde almacenar `data` y lo agrega al árbol. Si el árbol esta vacío, agrega el dato en un nuevo `n3treeElem_t` que tenga sus punteros `left` y `right` apuntando a cero, y `center` apuntando a una lista vacía. Si el árbol no es vacío, compara el dato pasado por parámetro con el del primer nodo. Si es menor lo agrega en su subárbol izquierdo, llamando recursivamente a la función. Si es mayor, lo hace en su subárbol derecho respectivamente. Si es igual, lo agrega a la lista `center` del nodo.

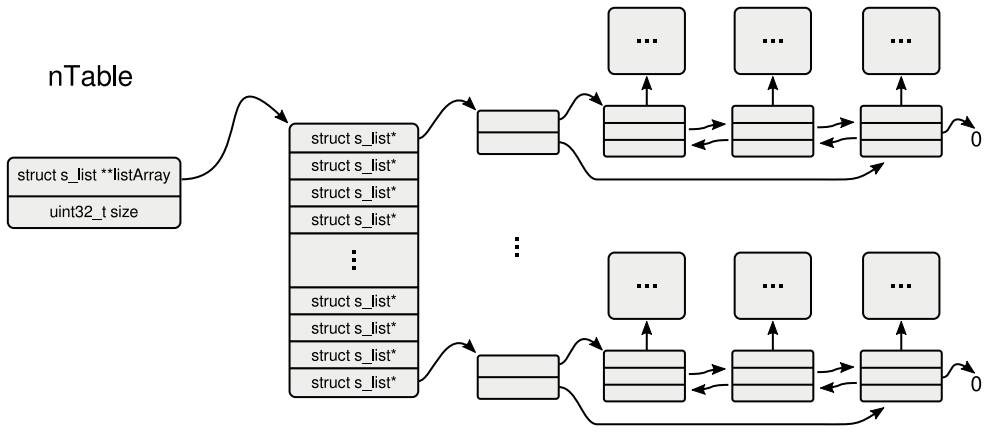
- `void n3treeRemoveEq(n3tree_t* t, funcDelete_t* fd)`
  Recorre recursivamente todo el árbol, borrando en cada nodo todos los elementos de su lista de datos iguales (`center`). Si `fd` no es cero, utiliza la función para borrar los datos respectivamente.

- `void n3treeDelete(n3tree_t* t, funcDelete_t* fd)`
  Borra todos los nodos del árbol, incluyendo a `n3tree_t`. Si el valor de `fd` no es cero, utiliza la función para borrar todos los datos.

- `void n3treePrint(n3tree_t* t, FILE *pFile, funcPrint_t* fp)`
  Escribe en el *stream* indicado por `pFile` el árbol almacenado en `t`. Para cada dato llama a la función `fp` y si su valor es cero, escribe el puntero al dato con el formato `"%p"`. Los nodos del árbol serán escritos respetando el orden: *In-order*[2] y siguiendo el formato: `<<` $x_0$`,...,`$x_{n-1}$ `>>`, donde $x_i$ es el resultado de escribir el i-ésimo dato. Si un nodo tiene su lista `center` no vacía, esta será escrita a continuación de la forma: ... $x_i$`[`$x_0$`,...,`$x_{n-1}$`]` ...

## Estructura `nTable`

La estructura `nTable_t` almacena un arreglo de listas de tamaño `size`, a partir del puntero `listArray`. Llamaremos a cada lista `slot`. Las funciones realizarán operaciones sobre algún `slot` seleccionado. Si el número de `slot` 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).



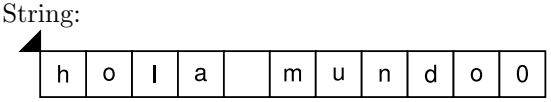

[2] `https://en.wikipedia.org/wiki/Tree_traversal#In-order_(LNR)`

```
typedef struct s_nTable{
    struct s_list **listArray;
    uint32_t size;
} nTable_t;
```

- `nTable_t* nTableNew(uint32_t size)`
  Crea un nuevo `nTable_t` con un arreglo de tamaño `size`, completándolo con listas vacías.
- `void nTableAdd(nTable_t* t, uint32_t slot, void* data, funcCmp_t* fc)`
  Agrega un nuevo elemento que contenga `data` a la lista indicada por el número de `slot`.
- `void nTableRemoveSlot(nTable_t* t, uint32_t slot, void* data, funcCmp_t* fc, funcDelete_t* fd)`
  Borra los elementos iguales a `data` de la lista indicada por el número de `slot`. Si `fd` no es cero, utiliza la función para borrar los datos en cuestión.
- `void nTableRemoveAll(nTable_t* t, void* data, funcCmp_t* fc, funcDelete_t* fd)`
  Borra todas las apariciones de `data` en cualquiera de las listas (cualquier `slot`). Si el valor de `fd` no es cero, utiliza la función para borrar los datos dados.
- `void nTableDeleteSlot(nTable_t* t, 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 nTableDelete(nTable_t* t, funcDelete_t* fd)`
  Borra todas las estructuras apuntadas desde `nTable_t`. Si `fd` no es cero, utiliza la función para borrar sus datos respectivamente.
- `void nTablePrint(nTable_t* t, 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: $i =$ `[`$x_0$`,...,`$x_{n-1}$`]`, donde $x_i$ es el resultado de escribir el i-ésimo dato.

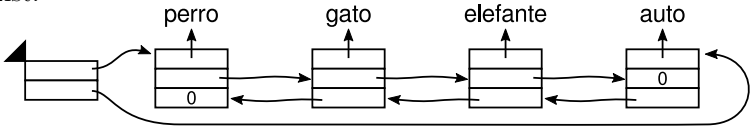
## Ejemplos

- String:



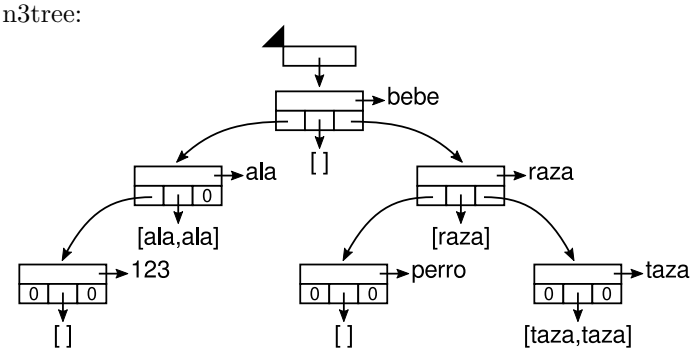

  Print: `hola mundo`

- List:



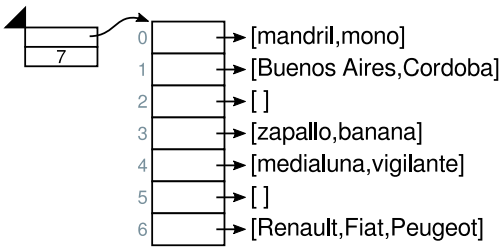

  Print: `[perro,gato,elefante,auto]`

- n3tree:




  Print: `<< 123 ala[ala,ala] bebe perro raza[raza] taza[taza,taza] >>`

- nTable:




Print:

```
0 = [mandril,mono]
1 = [Buenos Aires,Cordoba]
2 = []
3 = [zapallo,banana]
4 = [medialuna,vigilante]
5 = []
6 = [Renault,Fiat,Peugeot]
```

# 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.)
- `void listRemoveFirst(list_t* l, funcDelete_t* fd)`(16 Inst.)
- `void listRemoveLast(list_t* l, funcDelete_t* fd)`(16 Inst.)
- `void listRemove(list_t* l, void* data, funcCmp_t* fc, funcDelete_t* fd)`(49 Inst.)
- `void listDelete(list_t* l, funcDelete_t* fd)`(26 Inst.)
- `void listPrint(list_t* l, FILE *pFile, funcPrint_t* fp)`(40 Inst.)
- `n3tree_t* n3treeNew()`(6 Inst.)
- `void n3treeAdd(n3tree_t* t, void* data, funcCmp_t* fc)`(47 Inst.)
- `void n3treeRemoveEq(n3tree_t* t, funcDelete_t* fd)`(29 Inst.)
- `void n3treeDelete(n3tree_t* t, funcDelete_t* fd)`(33 Inst.)
- `nTable_t* nTableNew(uint32_t size)`(31 Inst.)
- `void nTableAdd(nTable_t* t, uint32_t slot, void* data, funcCmp_t* fc)`(9 Inst.)
- `void nTableRemoveSlot(nTable_t* t, uint32_t slot, void* data, funcCmp_t* fc, funcDelete_t* fd)`(11 Inst.)
- `void nTableDeleteSlot(nTable_t* t, uint32_t slot, funcDelete_t* fd)`(21 Inst.)
- `void nTableDelete(nTable_t* t, funcDelete_t* fd)`(21 Inst.)

Además las siguientes funciones deben ser implementadas en lenguaje C.

- `char* strRange(char* a, uint32_t i, uint32_t f)`(15 líneas)
- `void listPrintReverse(list_t* l, FILE *pFile, funcPrint_t* fp)`(15 líneas)
- `void n3treePrint(n3tree_t* t, FILE *pFile, funcPrint_t* fp)`(11 líneas)
- `void nTableRemoveAll(nTable_t* t, void* data, funcCmp_t* fc, funcDelete_t* fd)`(2 líneas)
- `void nTablePrint(nTable_t* t, 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:

1- **Caso n3tree**

- Crear un n3tree vacío.
- Agregar exactamente 10 strings cualquiera.
- Imprimir el n3tree.

2- **Caso nTable**

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

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 11/04. 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 (`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`.