# Trabajo Práctico 1
# La oración

Organización del Computador II

Segundo Cuatrimestre 2015

## 1. Introducción

El objetivo de este TP es implementar un conjunto de funciones sobre una estructura recursiva que representa una oración, o sea, una lista de palabras. La forma más común de implementar una **lista** es mediante la concatenación de **nodos**. El nodo define la forma y contenido de los elementos de la lista. Además, el nodo define cuál es el nodo que le sigue (*siguiente*). Así, identificando cuál es el primer nodo, una lista es un conjunto de nodos que se siguen unos a otros, del primero al último. A este tipo de lista se la denomina **simplemente enlazada**, o directamente **lista**. La cantidad de elementos de esta lista es variable.

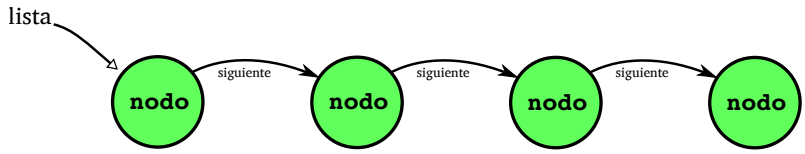


Figura 1: Ejemplo de Lista Simplemente Enlazada.

La mayor parte de los ejercicios a realizar para este TP estarán divididos en dos secciones:

- **Primera:** Completar las funciones que permiten manipular de forma básica una lista.
- **Segunda:** Realizar las funciones avanzadas sobre listas.

Por último se deberá realizar un programa en `lenguaje C`, que cree determinadas listas y ejecute algunas de las funciones de manipulación de las mismas.

### 1.1. Tipos lista y nodo

Para poder representar listas de distintas cosas los nodos suelen tener además del puntero al nodo siguiente, un puntero al dato que representa el contenido del nodo. En nuestro caso los nodos contendrán palabras representadas por *strings* o *char**.

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

typedef struct nodo_t {
    struct nodo_t *siguiente;
    char *palabra;
} __attribute__((__packed__)) nodo;
```

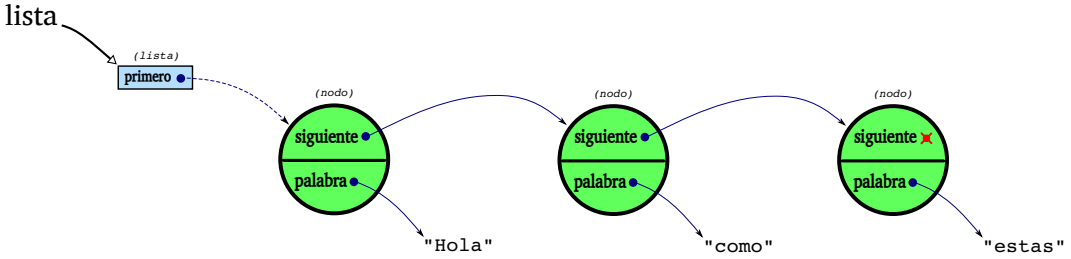


Figura 2: Ejemplo de una `lista` de palabras que representan una oración.

## 1.2. Funciones de palabras

- `unsigned char palabraLongitud( char *p )`
  Devuelve la cantidad de caracteres de `p`.

- `bool`[1] `palabraMenor( char *p1, char *p2 )`
  Indica si `p1` $<$ `p2` en orden lexicográfico, en función de los valores de codificación ASCII de cada caracter. Es decir, los caracteres en mayúsculas son menores a los caracteres en minúsculas, y los caracteres que representan los números son menores a todos los anteriores.

  Por ejemplo: **merc**ed $<$ **merc**urio, perro $<$ zorro, **senior** $<$ **senior**a, **ca**Za $<$ **ca**sa y **hola** $\not<$ **hola**

  **Nota:** La función auxiliar `palabraIgual` ya se encuentra implementada en `lenguaje C`, aprovéchela como un pseudocódigo para pensar en la implementación de `palabraMenor`.

- `void palabraFormatear( char *p, void (*funcModificarString)(char*) )`
  Modifica la palabra `p` aplicándole la función `funcModificarString` pasada por parámetro.

- `void palabraImprimir( char *p, FILE *file )`
  Escribe en el archivo `file` pasado por parámetro la palabra `p`. Al final de cada palabra se agregará el caracter especial de fin de línea (`LF = 10`).

- `char *palabraCopiar( char *p )`
  Devuelve una nueva copia de `p`.

  **Nota 1:** Si le sirve de ayuda, aproveche la función `palabraLongitud`.

  **Nota 2:** Nunca olvide las particularidades del formato de *strings* de *C*.

## 1.3. Funciones de lista y nodo

- `nodo *nodoCrear( char *palabra )`
  Crea un nodo con la `palabra` pasada por parámetro.

  **Nota:** No es necesario crear una nueva copia de la palabra.

- `void nodoBorrar( nodo *n )`
  Borra el nodo `n` y todo su contenido liberando toda la memoria utilizada por el nodo.

- `lista *oracionCrear( void )`
  Crea una lista vacía.

- `void oracionBorrar( lista *l )`
  Borra la lista y todos sus nodos internos.

  **Nota:** Piense cómo utilizar la función `nodoBorrar`.

- `void oracionImprimir( lista *l, char *archivo, void (*funcImprimirPalabra)(char*,FILE*) )`
  Agrega al `archivo` pasado por parámetro los datos de `l`. Estos datos deben imprimirse siguiendo el orden que tienen en la lista. Esta función no escribe directamente en el archivo, sino que lo hace aplicando la función `funcImprimirPalabra` a cada `palabra` de los nodos de `l`, escribiendo así los datos correctamente.
  Si la `lista` está vacía se deberá imprimir `<oracionVacia>` seguido del caracter especial de fin de línea (`LF = 10`).
  El archivo se debe abrir en modo `append`, de modo que las nuevas líneas sean adicionadas al archivo original.

  Por ejemplo, para la `lista` de la Figura 2, si hiciéramos

  ```
  oracionImprimir( l, "salida.txt", palabraImprimir )
  ```

  obtendríamos:

  *(...)*
  `Hola` *[LF]*
  `como` *[LF]*
  `estas` *[LF]*

[1] Utilizado para indicar un valor de verdad. Es un `byte` que vale 0 o 1 y está definido en `stdbool.h`

## 1.4. Funciones Avanzadas

- `float longitudMedia( lista *l )`
  Dada una lista `l` de palabras devuelve el promedio de sus longitudes.

  **Nota:** Si le sirve de ayuda, aproveche la función `palabraLongitud`.

- `void insertarOrdenado( lista *l, char *palabra, bool (*funcCompararPalabra)(char*,char*) )`
  Inserta la `palabra` en un nuevo `nodo` en `l`, manteniendo el orden dado por `funcCompararPalabra`. Suponiendo que la lista ya tiene sus `nodo`s ordenados de forma creciente según `funcCompararPalabra` sobre cada `palabra`, el nuevo nodo se insertará respetando ese orden. Se considera que una lista vacía está ordenada.

  **Nota 1:** Si le sirve de ayuda, aproveche las funciones `nodoCrear` e `insertarAtras` para algún caso particular. La primera ya implementada anteriormente, y la segunda ya implementada en `lenguaje C`.

  **Nota 2:** Al finalizar esta función, tanto la lista como todos sus nodos, deben respetar el invariante de la estructura `lista`. Tenga en cuenta la aritmética de punteros de todas las estructuras.

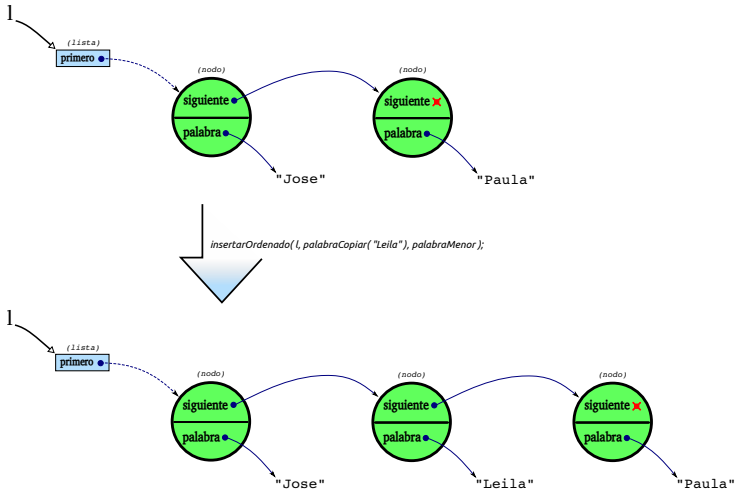


Figura 3: Ejemplo de aplicar `insertarOrdenado`.

- `void filtrarPalabra( lista *l, bool (*funcCompararPalabra)(char*,char*), char *palabraCmp )`
  Dada una lista de palabras `l`, una función de comparación `funcCompararPalabra` y una palabra de comparación `palabraCmp`, modifica la lista tal que sólo queden los nodos de `l` que cumplen `funcCompararPalabra( palabraActual, palabraCmp )`, donde `palabraActual` es la *palabra* de algún `nodo` de `l`. Los nodos que no cumplen deben ser eliminados.
  Los elementos de la lista resultante estarán en el mismo orden relativo en el que se encontraban originalmente.

  **Nota 1:** Si le sirve de ayuda, aproveche la función `nodoBorrar` ya implementada anteriormente.

  **Nota 2:** Al finalizar esta función, tanto la lista como todos sus nodos, deben respetar el invariante de la estructura `lista`. Tenga en cuenta la aritmética de punteros de todas las estructuras.

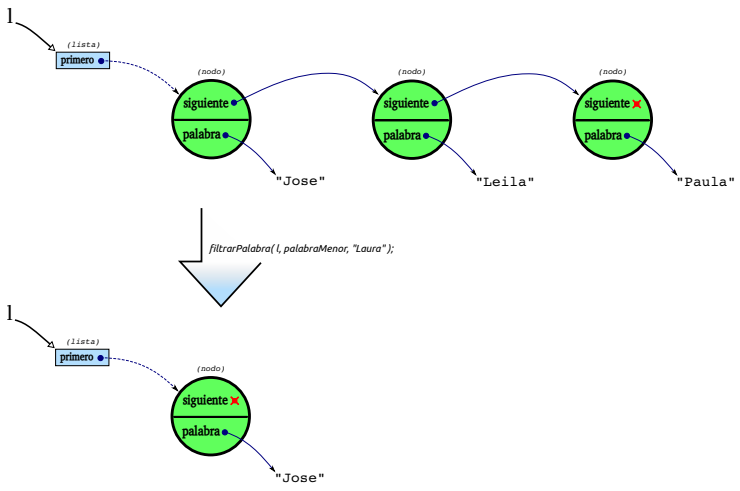


Figura 4: Ejemplo de aplicar `filtrarAltaLista`.

- `void descifrarMensajeDiabolico( lista *l, char *archivo, void (*funcImpPbr)(char*,FILE*) )`
  Descifra el posible mensaje diabólico escondido en la oración al ser leída al revés[2]. Para cumplir este objetivo, la función agrega al `archivo` pasado por parámetro los datos de `l`. Estos datos deben imprimirse siguiendo el orden inverso que tienen en la lista, es decir, desde el último hacia el primero. Esta función no escribe directamente en el archivo, sino que lo hace aplicando la función `funcImpPbr` a cada `palabra` de los nodos de `l`, escribiendo así los datos correctamente.
  Si la `lista` está vacía se deberá imprimir `<sinMensajeDiabolico>` seguido del caracter especial de fin de línea (`LF = 10`).
  El archivo se debe abrir en modo `append`, de modo que las nuevas líneas sean adicionadas al archivo original. Por ejemplo, para la `lista` de la Figura 2, si hiciéramos

```
descifrarMensajeDiabolico( l, "salida.txt", palabraImprimir )
```

  obtendríamos:

  *(...)*
  `estas` *[LF]*
  `como` *[LF]*
  `Hola` *[LF]*

  **Nota:** Piense cómo utilizar la pila para realizar el recorrido inverso de las palabras.

## 1.5. Funciones ya implementadas en `lenguaje C`

- `bool palabraIgual( char *p1, char *p2 )`
  Indica si `p1` y `p2` son iguales, caracter a caracter, en función de los valores de codificación ASCII.
- `void insertarAtras( lista *l, char *palabra )`
  Inserta la `palabra` en un nuevo `nodo` al final de la lista.

## Importante:

a) Puede asumir que los parametros de tipo puntero siempre serán no nulos ($\neq$ `NULL`).

b) Puede asumir que todos los strings sólo contendrán los siguientes caracteres:

- números = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' }
- letras minúsculas = { 'a','b','c','d','e','f','g', 'h','i','j','k','l','m','n', 'o','p','q','r','s','t','u', 'v','w','x','y','z' }
- letras mayúsculas = { 'A','B','C','D','E','F','G', 'H','I','J','K','L','M','N', 'O','P','Q','R','S','T','U', 'V','W','X','Y','Z' }

[2] Existe una teoría de que los mensajes subliminales pueden percibirse al ser leídos o escuchados en forma invertida. El efecto es conocido como *backmasking* o enmascaramiento hacia atrás. Una gran cantidad de artistas fueron acusados de escribir canciones con mensajes satánicos: Led Zeppelin, Queen, Los Beatles, Rollings Stones, Black Sabath, Xuxa, Ricky Martin, Diego Torres y hasta Shakira!

# 2. Enunciado

## 2.1. Las funciones a implementar en lenguaje ensamblador de 64 bits son:

- `unsigned char palabraLongitud( char *p )` *[15]*
- `bool palabraMenor( char *p1, char *p2 )` *[30]*
- `void palabraFormatear( char *p, void (*funcModificarString)(char*) )` *[5]*
- `void palabraImprimir( char *p, FILE *file )` *[15]*
- `char *palabraCopiar( char *p )` *[25]*
- `nodo *nodoCrear( char *palabra )` *[15]*
- `void nodoBorrar( nodo *n )` *[15]*
- `lista *oracionCrear( void )` *[10]*
- `void oracionBorrar( lista *l )` *[20]*
- `void oracionImprimir( lista *l, char *archivo, void (*funcImprimirPalabra)(char*,FILE*) )` *[35]*
- `float longitudMedia( lista *l )` *[30]*
- `void insertarOrdenado( lista *l, char *palabra, bool (*funcCompararPalabra)(char*,char*) )` *[35]*
- `void filtrarPalabra( lista *l, bool (*funcCompararPalabra)(char*,char*), char *palabraCmp )` *[35]*
- `void descifrarMensajeDiabolico( lista *l, char *archivo, void (*funcImpPbr)(char*,FILE* ) )` *[45]*

## Importante:

Todas las funciones auxiliares que decida agregar deberán estar implementadas en `lenguaje ensamblador` al momento de la entrega del TP.

## Nota:

En cada función a implementar se indica, con la referencia *[N]*, la cantidad aproximada de líneas de código que fueron necesarias para resolver la misma según la solución de la cátedra. Su utilidad es tener una idea del tamaño relativo que tienen las distintas funciones. De ninguna manera es necesario realizar implementaciones de las funciones que cumplan o superen esa cota.

## 2.2. Compilación y Testeo

Para compilar el código y poder correr las pruebas cortas deberá ejecutar `make main` y luego `./pruebacorta.sh`. Para compilar el código y correr las pruebas intensivas deberá ejecutar `./prueba.sh`.

### 2.2.1. Pruebas cortas

Deberá construirse un programa de prueba (`main.c`) que realice las acciones detalladas a continuación. La idea del ejercicio es verificar incrementalmente que las funciones que se vayan implementando funcionen correctamente. El programa puede correrse con `./pruebacorta.sh` para verificar que no se pierde memoria ni se realizan accesos incorrectos a la misma. Recordar siempre borrar las listas luego de usarlas y todas aquellas instancias que cree en memoria dinámica.

1- **Calcular la longitud de una palabra.** Para esto deberá implementar la función `palabraLongitud`. Luego de implementarla, es muy útil convencerse de que toda marcha bien, por ejemplo, calculando la longitud de varias palabras distintas, una palabra vacía, y mostrando el resultado por consola. Podría hacer:

```
printf( "la longitud de 'hola' es = %d \n", palabraLongitud( "hola" ) );
```

2- **Comparar una palabra con otra.** Para esto deberá implementar la función `palabraMenor`. Para estar completamente seguro que todo anda bien, pruebe estas funciones con los ejemplos del enunciado: **merc**ed < **merc**urio, perro < zorro, **senior** < **senior**a, **ca**Za < **ca**sa y **hola** $\not<$ **hola** y corrobore que el resultado sea el esperado. Por ejemplo, muestre el resultado por consola:

```
if( palabraMenor( "merced", "mercurio" ) )
    printf( "TRUE\n" ); else printf( "FALSE\n" );
```

3- **Aplicarle formato a una palabra.** Para esto deberá implementar la función `palabraFormatear`. Ya que esta función necesita de una función de strings, se puede probar con una función que no haga nada, por ejemplo, definiendo `f` de la siguiente manera:

```
void f( char* s ){}
char unaPalabra[] = "hola";
palabraFormatear( unaPalabra, f );
```

Si quiere estar completamente seguro del correcto funcionamiento de `palabraFormatear`, puede implementar rápidamente una función que modifique arbitrariamente algún caracter del string y observar el resultado antes y después de aplicarle la función, por ejemplo, definiendo `g` de la siguiente manera:

```
void g( char *s ){ if( s[0] != 0 ) s[0] = 'X'; }
palabraFormatear( unaPalabra, g );
```

4- **Imprimir una palabra.** Para esto deberá implementar la función `palabraImprimir`. Ya que esta función necesita un `FILE*`, para probar la función momentáneamente se puede pasarle como parámetro a la propia consola de la siguiente manera:

```
estudianteImprimir( unaPalabra, stdout );
```

5- **Copiar una palabra.** Para esto deberá implementar la función `palabraCopiar`. Para facilitarse esta tarea puede utilizar la función `palabraLongitud` ya implementada. Pruebe copiando un string, modificandolo, imprimiendo por consola el original y la nueva copia y verificando que el resultado sea el esperado. Por ejemplo:

```
char *unaPalabra  = palabraCopiar( "hola" );
char *otraPalabra = palabraCopiar( unaPalabra );
unaPalabra[1] = 'X';
palabraImprimir( unaPalabra, stdout );
palabraImprimir( otraPalabra, stdout );
free( unaPalabra );
free( otraPalabra );
```

6- **Crear un nodo y borrarlo.** Para esto deberá implementar las funciones `nodoCrear` y `nodoBorrar`. La función `nodoCrear` recibe una *palabra* como parámetro, pruebe pasándole alguna palabra como parámetro, por ejemplo, de la siguiente manera:

```
nodo *miNodo = nodoCrear( palabraCopiar( "algunaPalabra" ) );
```

Luego pruebe borrar el nodo, por ejemplo, de la siguiente manera:

```
nodoBorrar( miNodo );
```

Luego de esto, suele ser muy útil convencerse de que toda marcha bien, por ejemplo, mostrando "a mano" el contenido del nodo recién creado. Podría hacer:

```
printf( "Palabra del Nodo: %s\n", miNodo->palabra );
```

7- **Crear una lista vacía, imprimirla y borrarla.** Deberá implementar `oracionCrear`, `oracionBorrar` y `oracionImprimir`. No es necesario implementar completamente `oracionImprimir` y `oracionBorrar`. Ya que se trabaja con una lista vacía, puede implementar únicamente este caso en ambas funciones y dejar el resto de los casos para el siguiente punto. Ya que la función `oracionImprimir` necesita de una función para saber cómo imprimir las palabras, puede utilizar la función `palabraImprimir` ya implementada, por ejemplo, de la siguiente manera:

```
lista *miLista = oracionCrear();
oracionImprimir( miLista, "salida.txt", palabraImprimir );
oracionBorrar( miLista );
```

Puede serle útil, para verificar manualmente que la salida es correcta, llamar a la función `oracionImprimir` con el argumento `"/dev/stdout"` como nombre de archivo. En los sistemas operativos Unix modernos escribir en este archivo equivale a escribir en la salida estándar del proceso.

8- **Crear una lista nueva, agregarle una palabra, imprimirla y borrarla.** Ahora sí debe completar las funciones `oracionImprimir` y `oracionBorrar`. A esta altura, no es necesario implementar `insertarOrdenado` para agregar una palabra, puede simplemente utilizar `insertarAtras` que ya está implementada en lenguaje `C` para probar insertando las palabras siempre al final de la lista, ya que aún no nos importa su orden en la misma. Pruebe nuevamente el correcto funcionamiento de estas funciones, por ejemplo, de la siguiente manera:

```
lista *miLista = oracionCrear();
insertarAtras( miLista, palabraCopiar( "palabra1" ) );
oracionImprimir( miLista, "salida.txt", palabraImprimir );
oracionBorrar( miLista );
```

Pruebe insertar varias palabras, imprimir la lista y corrobore que los resultados sean los esperados.

9- **Calcular la longitud media de la oración.** Deberá implementar `longitudMedia`. Pruebe el correcto funcionamiento de la misma, por ejemplo, de la siguiente manera:

```
lista *miLista = oracionCrear();
insertarAtras( miLista, palabraCopiar( "palabra1" ) );
printf( "LongMedia = %2.5f\n", longitudMedia( miLista ) );
```

Pruebe calcular la longitud media em varios casos, una lista vacía, con una palabra, con dos, etc., y corrobore que los resultados sean los esperados.

10- **Crear una lista, agregarle palabras de forma ordenada, imprimirla y borrarla.** Ahora sí debe implementar `insertarOrdenado`. Piense detenidamente todos los casos bordes de inserción de nodos que debe tener en cuenta e impleméntelos. Ya que la función `insertarOrdenado` necesita de una función para saber cómo comparar las palabras, puede utilizar la función `palabraMenor` ya implementada, por ejemplo, de la siguiente manera:

```
lista *miLista = oracionCrear();
insertarOrdenado( miLista, palabraCopiar( "palabra1" ), palabraMenor );
oracionImprimir( miLista, "salida.txt", palabraImprimir );
```

Pruebe recrear el ejemplo de la Figura 3. Pruebe insertar varias palabras de modo que se fuercen los distintos casos bordes implementados en la función. Corrobore que los resultados sean los esperados.

11- **Corroborar el invariante de la lista.** Si está seguro que conservó y respetó el invariante de la estructura `lista` al finalizar cada una de todas las funciones anteriores, puede omitir este punto. Si no, puede aprovechar a chequearlo en este momento. Para este tipo de estructuras simples en general alcanza con recorrer de principio a fin la lista. Una forma fácil de hacerlo es utilzando la función `oracionImprimir` ya implementada. Puede utilizarla para imprimir la lista cada vez que la modifica con alguna nueva función implementada, ver el resultado y compararlo con el esperado.

12- **Implementar `filtrarPalabra`.** Piense detenidamente todos los casos bordes de eliminación de nodos que debe tener en cuenta e impleméntelos. Ya que la función `filtrarPalabra` necesita de una función para saber cómo comparar las palabras, puede utilizar las funciones `palabraMenor` y `palabraIgual`, por ejemplo, de la siguiente manera:

```
lista *miLista = oracionCrear();
insertarOrdenado( miLista, palabraCopiar( "palabra1" ), palabraMenor );
insertarOrdenado( miLista, palabraCopiar( "palabra2" ), palabraMenor );
insertarOrdenado( miLista, palabraCopiar( "palabra3" ), palabraMenor );
filtrarPalabra( miLista, palabraMenor, "palabra3" );
oracionImprimir( miLista, "salida.txt", palabraImprimir );
filtrarPalabra( miLista, palabraIgual, "palabra1" );
oracionImprimir( miLista, "salida.txt", palabraImprimir );
```

Pruebe recrear el ejemplo de la Figura 4. Pruebe casos con varias palabras de modo que se fuercen los distintos casos bordes implementados en la función. Corrobore que los resultados sean los esperados.

13- **Implementar `descifrarMensajeDiabolico`.** Piense detenidamente cómo utilizar la pila para realizar un recorrido inverso de las palabras que están en la lista y luego impleméntela. Ya que la función necesita de una función para saber cómo imprimir las palabras, puede utilizar la función `palabraImprimir` ya implementada, por ejemplo, de la siguiente manera:

```
descifrarMensajeDiabolico( miLista, "salida.txt", palabraImprimir );
```

Pruebe con varios ejemplos y todos los casos bordes se le ocurran. Corrobore que los resultados sean los esperados.

### 2.2.2. Pruebas intensivas (Testing)

En un ataque de bondad, hemos decidido proveer 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 `./prueba.sh`, que compilará el *tester* y correrá todos los tests de la cátedra. Un test consiste en la creación, inserción, eliminación y ejecución de funciones avanzadas e impresión en archivo de una gran cantidad de listas. 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.

## 2.3. Archivos

Se entregan los siguientes archivos:

- `lista.h`: Contiene la definición de las estructuras y de las funciones a realizar.
- `lista_asm.asm`: Archivo a completar con su código en `lenguaje ensamblador`.
- `lista_c.c`: Archivo con la implementación de las funciones ya implementadas en `lenguaje C`. Aquí también puede realizar las primeras implementaciones de sus funciones en `lenguaje C` para luego utilizarlas como pseudocódigo para implementarlas en `lenguaje ensamblador` en `lista_asm.asm`.
- `Makefile`: Contiene las instrucciones para ensamblar y compilar el código.
- `main.c`: Es el archivo principal donde escribir los ejercicios para las pruebas cortas (`pruebacorta.sh`).
- `tester.c`: Es el archivo del tester de la cátedra. No debe ser modificado.
- `pruebacorta.sh`: Es el script que corre el test simple (pruebas cortas). No debe ser modificado.
- `prueba.sh`: Es el script que corre todos los test intensivos. No debe ser modificado.
- `salida.caso*.catedra.txt`: Archivos de salida que se compararán con sus salidas. No modificarlos.

**Notas:**

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

b) Para el manejo de archivos se recomienda usar las funciones de `C`: `fopen`, `fprintf` y `fclose`.

c) Para el manejo de strings **no está permitido** usar las funciones de `C`: `strlen`, `strcpy`, `strcmp` y `strdup`.

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. Es condición necesaria para la aprobación que el TP supere correctamente todos los tests.

# 3. Resolución, Informe y Forma de Entrega

Este TP es de carácter **individual**. No deberán entregar un informe. En cambio, deberán entregar un archivo comprimido que contenga únicamente los archivos `lista_asm.asm` y `main.c`.

**Este TP deberá entregarse como máximo a las 16:59:59 hs del día Martes 8 de Septiembre de 2015.** ***La reentrega del TP, en caso de ser necesaria, deberá realizarse como máximo a las 16:59:59 hs del día Martes 29 de Septiembre de 2015.***

Ambas entregas se realizarán a través de la página web. El sistema sólo aceptará entregas de TPs hasta el horario de entrega, sin excepciones.

Ante cualquier problema con la entrega, comunicarse por mail a la lista de docentes.