Trabajo Práctico 1
Alta Lista

Organización del Computador II

Primer Cuatrimestre 2015

1. Introducción

El objetivo de este TP es implementar un conjunto de funciones sobre una estructura recursiva que representa una lista de estudiantes y que forma parte de *El Sistema* utilizado para la administración de los grupos de estudiantes de Orga2 para el TP2 y TP3.

Este sistema ya viene funcionando desde cuatrimestres anteriores, pero algunas de sus funcionalidades fueron presentando algunos problemas. Por eso, para poder encarar correctamente la administración de los grupos de TPs venideros, los estudiantes de Orga2 realizarán implementaciones propias de algunas de las funcionalidades de *El Sistema* para actualizar algunos de sus módulos antes de que deban armarse los grupos para el TP2.

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*). Y en algunas estructuras más complejas, que permiten realizar operación más avanzadas sobre las listas, el nodo también define cuál es el nodo que le antecede (*anterior*). Así, identificando cuál es el primer y último nodo, una lista es un conjunto de nodos que se siguen unos a otros, del primero al último y del último al primero. A este tipo de lista se la denomina **doblemente enlazada**. La cantidad de elementos de esta lista es variable.

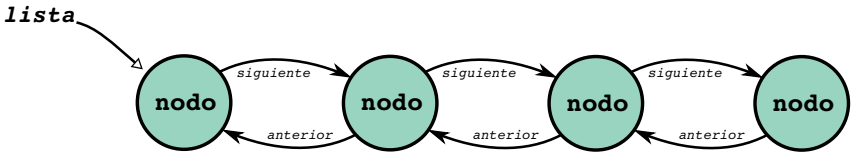

Figura 1: Ejemplo de Lista Doblemente Enlazada.

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 altaLista, nodo y estudiante

Para poder representar listas de distintos tipos, que contengan distintas "cosas", los nodos suelen tener además de los punteros al nodo anterior y el siguiente, un puntero a *void* representando el contenido (*dato*) de cada nodo. Esto hace que la lista sea "genérica". En este TP la llamaremos `altaLista`.

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

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

Y como en este caso queremos representar listas de estudiantes, utilizaremos para el contenido de los nodos la siguiente estructura que representa a los estudiantes. Un estudiante contiene: *nombre*, *grupo* y *edad*. Y está definido de la siguiente manera.

```
typedef struct estudiante_t {
    char            *nombre;
    char            *grupo;
    unsigned int    edad; //
} __attribute__((__packed__)) estudiante;
```

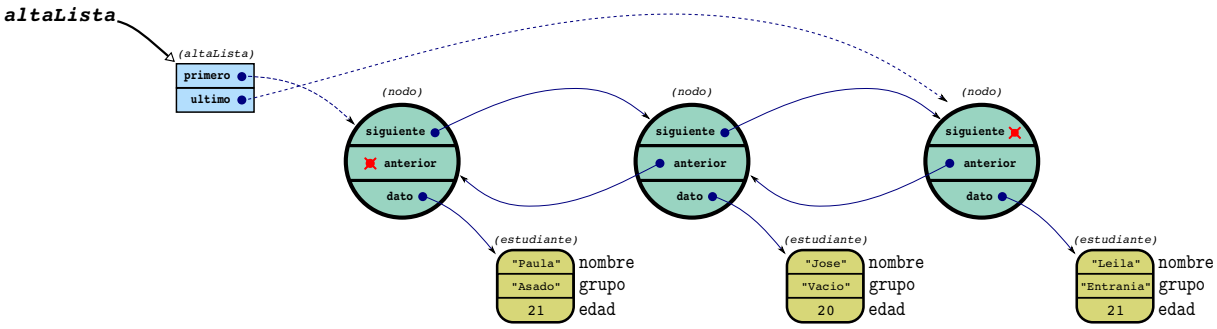

Figura 2: Ejemplo de una `altaLista` de `Estudiantes`.

1.2. Definiciones de punteros a funciones

Al ser una lista genérica, es necesario contar además con definiciones de funciones que especifiquen cómo se borra un dato, cómo se imprime, y cómo se compara con otro dato. Para este objetivo, se encuentran las siguientes definiciones de *punteros a funciones* que se utilizarán en las funciones de `altaLista`, `nodo` y `estudiante`:

- `typedef void (*tipoFuncionBorrarDato) (void*)`
  tipoFuncionBorrarDato toma un `dato` (tipo `void*`) y lo borra.
- `typedef void (*tipoFuncionImprimirDato) (void*, FILE*)`
  tipoFuncionImprimirDato toma un `dato` (tipo `void*`) y un puntero a un archivo (tipo `FILE*` [1]) e imprime el `dato` en ese archivo.
- `typedef bool (*tipoFuncionCompararDato) (void*, void*)`
  tipoFuncionCompararDato toma dos `datos` (tipo `void*`) y determina si la comparación es cierta entre ambos `datos` con algún cierto criterio de comparación.
- `typedef void (*tipoFuncionModificarString) (char*)`
  tipoFuncionModificarString toma un string y lo modifica.

[1] típico retorno de la función `fopen` como un puntero al buffer en memoria con los datos del archivo abierto.

1.3. Funciones de estudiante

- `estudiante *estudianteCrear( char *nombre, char *grupo, unsigned int edad )`
  Crea un estudiante con los datos pasados por parámetro. Los strings *nombre* y *grupo* pasados por parámetro deben ser copiados en el nuevo estudiante.

  Nota: Piense cómo facilitarse esta tarea utilizando la función auxiliar sugerida `string_copiar`.

- `void estudianteBorrar( estudiante *e )`
  Borra el estudiante pasado por parámetro.

  Nota: No olvide borrar todo el contenido del estudiante.

- `bool`[2] `menorEstudiante( estudiante *e1, estudiante *e2 )`
  Indica si `e1` es menor a `e2`. Se considera que `e1` es menor a `e2`, si el string `nombre` de `e1` es menor al string `nombre` de `e2`. Si el string `nombre` de ambos es igual, entonces es menor si la edad de `e1` es menor o igual a la de `e2`.

  Nota 1: Al comparar strings utilizamos la relación de orden típica para comparar cadenas de caracteres pero en función de los valores de codificación ASCII de cada caracter. Es decir que, de esta forma, 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 2: Piense cómo facilitarse esta tarea utilizando la función auxiliar `string_iguales` y la función auxiliar sugerida `string_menor`. La primera función ya se encuentra implementada en `lenguaje C`, puede utilizarla y aprovecharla además como un pseudocódigo para pensar en la implementación de `string_menor`.

- `void estudianteConFormato( estudiante *e, tipoFuncionModificarString f )`
  Modifica el `nombre` y `grupo` del estudiante aplicándoles la función `f` pasada por parámetro.

- `void estudianteImprimir( estudiante *e, FILE *file )`
  Escribe en el archivo `file` pasado por parámetro los datos de `e`.
  Cada dato (*nombre*, *grupo*, *edad*) del estudiante ocupará una nueva línea en el archivo. Para esto, al final de cada dato se agregará el caracter especial de fin de línea (`LF = 10`) . Los datos *grupo* y *edad* estarán precedidos del caracter especial de tabulación (`HT = 9`).
  Por ejemplo, para la primer estudiante de la `altaLista` de la Figura 2:

  (...)
  `Paula` *[LF]*
  [HT] `Asado` *[LF]*
  [HT] `21` *[LF]*

1.4. Funciones de altaLista y nodo

- `nodo *nodoCrear( void *dato )`
  Crea un nodo con el `dato` pasado por parámetro.

  Nota: No es necesario crear una nueva copia del dato.

- `void nodoBorrar( nodo *n, tipoFuncionBorrarDato f )`
  Borra el nodo `n` y su `dato` utilizando `f`.

- `altaLista *altaListaCrear( void )`
  Crea una altaLista vacía.

- `void altaListaBorrar( altaLista *l, tipoFuncionBorrarDato f )`
  Borra la lista y todos sus nodos internos, incluyendo el `dato` de cada uno utilizando `f`.

  Nota: Piense cómo utilizar la función `nodoBorrar`.

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

- `void altaListaImprimir( altaLista *l, char *archivo, tipoFuncionImprimirDato f )`
  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 `f` a cada `dato` de los nodos de `l`, escribiendo así los datos correctamente.
  Si la `altaLista` está vacía se deberá imprimir `<vacia>` 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 `altaLista` de la Figura 2, si hiciéramos

  ```
  altaListaImprimir( l, "salida.txt", (tipoFuncionImprimirDato)estudianteImprimir )
  ```

  obtendríamos:

  (...)
  `Paula` *[LF]*
  [HT] `Asado` *[LF]*
  [HT] `21` *[LF]*
  `Jose` *[LF]*
  [HT] `Vacio` *[LF]*
  [HT] `20` *[LF]*
  `Leila` *[LF]*
  [HT] `Entrania` *[LF]*
  [HT] `21` *[LF]*

1.5. Funciones Avanzadas

- `float edadMedia( altaLista *l )`
  Dada una lista de estudiantes `l`, devuelve la media de sus edades.

  Nota: No confundir *media*[3] con *mediana*[4].

- `void insertarOrdenado( altaLista *l, void *dato, tipoFuncionCompararDato f )`
  Inserta el `dato` en un nuevo `nodo` en `l`, manteniendo el orden dado por `f`. Suponiendo que la lista ya tiene sus `nodo`s ordenados de forma creciente según `f` sobre cada `dato`, el nuevo nodo se insertará respetando ese orden. Se considera que una lista vacía está ordenada.
  `f(dato1,dato2)` devuelve *true* si *dato1* $<$ *dato2*.

  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 `altaLista`. Tenga en cuenta la aritmética de punteros de todas las estructuras.

- `void filtrarAltaLista( altaLista *l, tipoFuncionCompararDato f, void *datoCmp )`
  Dada una lista de estudiantes `l`, una función de comparación `f` y un dato de comparación `datoCmp`, modifica la lista tal que sólo queden los nodos de `l` que cumplen `f( datoActual, datoCmp )`, donde `datoActual` es el *dato* 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 `altaLista`. Tenga en cuenta la aritmética de punteros de todas las estructuras.

[3] Media: `http://es.wikipedia.org/wiki/Media_aritm%C3%A9tica`
[4] Mediana: `http://es.wikipedia.org/wiki/Mediana_(estad%C3%ADstica)`

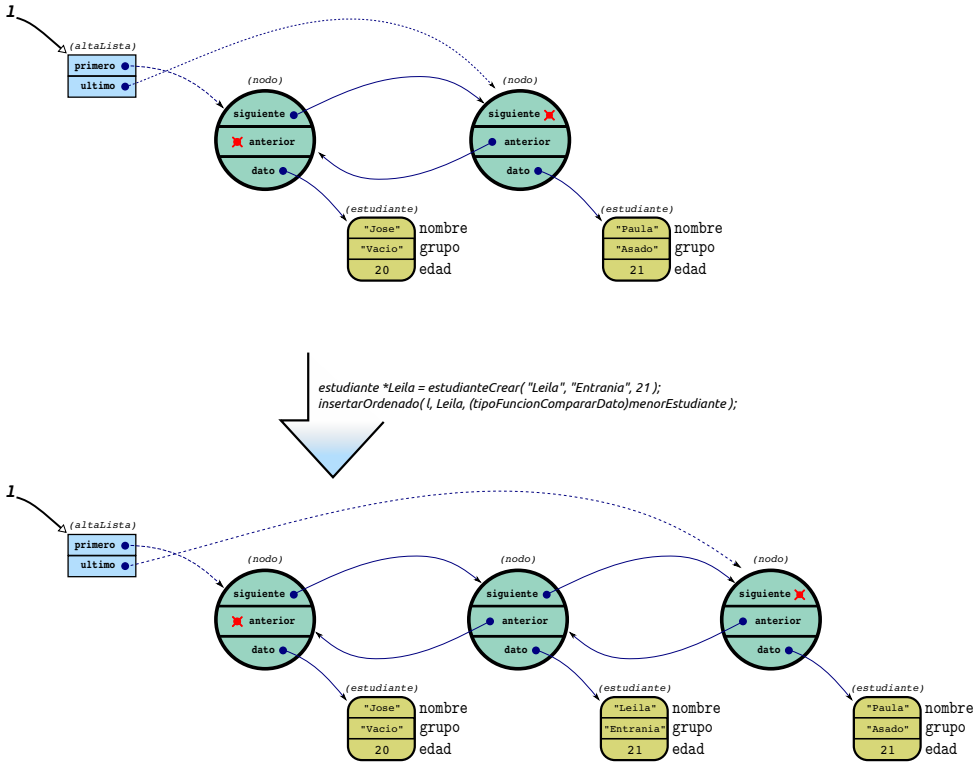

Figura 3: Ejemplo de aplicar `insertarOrdenado`.

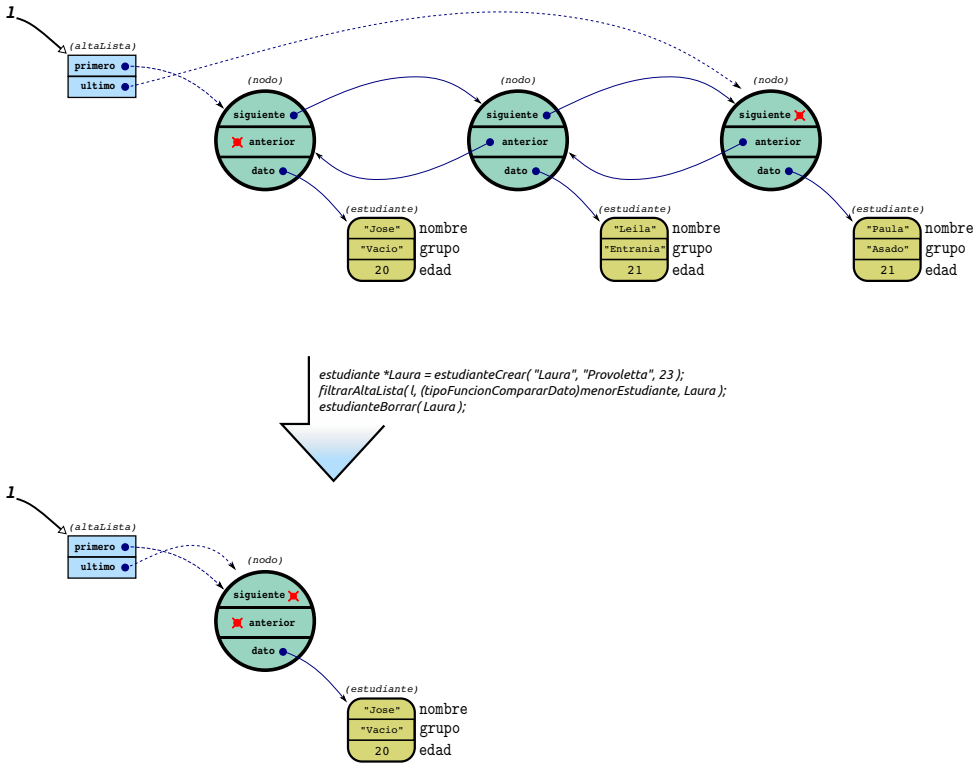

Figura 4: Ejemplo de aplicar `filtrarAltaLista`.

1.6. Funciones Auxiliares Sugeridas

- `unsigned char string_longitud( char *s )`
  Devuelve la cantidad de caracteres de `s`.

- `char *string_copiar( char *s )`
  Devuelve una nueva copia de `s`.

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

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

- `bool string_menor( char *s1, char *s2 )`
  Indica si `s1` $<$ `s2`, utilizamos la relación de orden típica para comparar cadenas de caracteres pero en función de los valores de codificación ASCII de cada caracter. Es decir que, de esta forma, 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 `string_iguales` ya se encuentra implementada en `lenguaje C`, aprovéchela como un pseudocódigo para pensar en la implementación de `string_menor`.

1.7. Funciones ya implementadas en `lenguaje C`

- `bool string_iguales( char *s1, char *s2 )`
  Indica si `s1` y `s2` son iguales, caracter a caracter, en función de los valores de codificación ASCII.

- `void insertarAtras( altaLista *l, void *dato )`
  Inserta el `dato` en un nuevo `nodo` al final de la lista.

Importante:

Puede asumir que los strings `nombre` y `grupo` de los estudiantes sólo contienen 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. Enunciado

2.1. Las funciones a implementar en lenguaje ensamblador son:

- `estudiante *estudianteCrear( char *nombre, char *grupo, unsigned int edad )` *[30]*
- `void estudianteBorrar( estudiante *e )` *[15]*
- `bool menorEstudiante( estudiante *e1, estudiante *e2 )` *[35]*
- `void estudianteConFormato( estudiante *e, tipoFuncionModificarString f )` *[20]*
- `void estudianteImprimir( estudiante *e, FILE *file )` *[20]*
- `nodo *nodoCrear( void *dato )` *[15]*
- `void nodoBorrar( nodo *n, tipoFuncionBorrarDato f )` *[15]*
- `altaLista *altaListaCrear( void )` *[10]*
- `void altaListaBorrar( altaLista *l, tipoFuncionBorrarDato f )` *[35]*
- `void altaListaImprimir( altaLista *l, char *archivo, tipoFuncionImprimirDato f )` *[40]*
- `float edadMedia( altaLista *l )` *[25]*
- `void insertarOrdenado( altaLista *l, void *dato, tipoFuncionCompararDato f )` *[60]*
- `void filtrarAltaLista( altaLista *l, tipoFuncionCompararDato f, void *datoCmp )` *[60]*

Las funciones auxiliares sugeridas

Para implementar todas las funciones anteriores de una manera concisa y fácil de entender se pueden implementar y utilizar también las funciones auxiliares sugeridas. Las mismas son opcionales. Queda a criterio de cada uno cómo utilizarlas para implementar las funciones anteriores y agregar todas las que considere.

- `unsigned char string_longitud( char *s )` *[15]*
- `char *string_copiar( char *s )` *[30]*
- `bool string_menor( char *s1, char *s2 )` *[25]*

Importante:

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

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- **Crear un estudiante y borrarlo.** Para esto deberá implementar las funciones `estudianteCrear` y `estudianteBorrar`.

   a) Para crear un estudiante es necesario hacer copias de los strings *nombre* y *grupo* pasados por parámetro. Para facilitarse esta tarea puede implementar las funciones auxiliares sugeridas `string_longitud` y `string_copiar`. Puede implementarla y probarla. Copiar un string, modificarlo, imprimir por consola el original y la nueva copia y verificar que el resultado sea el esperado.

   b) Luego de implementar `estudianteCrear` y `estudianteBorrar` es muy útil convencerse de que toda marcha bien, por ejemplo, mostrando "a mano" el contenido del estudiante recién creado. Si el estudiante recién creado se llamara `miEstudiante`, podría hacer:

```
printf( "Nombre: %s\n", miEstudiante->nombre );
```

   De igual forma puede hacer con el resto del contenido del estudiante.

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

```
estudianteImprimir( miEstudiante, stdout );
```

3- **Comparar un estudiante con otro.** Para esto deberá implementar la función `menorEstudiante`.

   a) Para comparar estudiantes es necesario comparar strings. Para facilitarse esta tarea puede utilizar la función ya implmentada `string_iguales` e implementar la función auxiliar sugerida `string_menor`. Por ejemplo, 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.

   b) Luego implemente la función `menorEstudiante` y corrobore su correcto funcionamiento. Por ejemplo, cree dos estudiantes, compárelos y muestre el resultado por consola:

```
if( menorEstudiante( miEstudiante1, miEstudiante2 ) )
    printf( "TRUE\n" ); else printf( "FALSE\n" );
```

   Pruebe todos los casos que se le ocurran, incluso aquellos casos bordes donde los estudiantes tienen igual nombre.

4- **Aplicarle formato a un estudiante.** Para esto deberá implementar la función `estudianteConFormato`. Ya que `estudianteConFormato` 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 ){}
estudianteConFormato( miEstudiante, f );
```

   Si quiere estar completamente seguro del correcto funcionamiento de `estudianteConFormato`, 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'; }
estudianteConFormato( miEstudiante, g );
```

5- **Crear un nodo y borrarlo.** Para esto deberá implementar las funciones `nodoCrear` y `nodoBorrar`. La función `nodoCrear` recibe un *dato* como parámetro, pruebe pasándole como parámetro un estudiante. Por otro lado, la función `nodoBorrar` necesita de una función para saber cómo borrar el *dato*, por lo que puede utilizar la función `estudianteBorrar` ya implementada realizando el *casteo* (conversión de tipos) correspondiente, por ejemplo, de la siguiente manera:

```
nodo *miNodo = nodoCrear( miEstudiante );
nodoBorrar( miNodo, (tipoFuncionBorrarDato)estudianteBorrar );
```

6- **Crear una altaLista vacía, imprimirla y borrarla.** Deberá implementar `altaListaCrear`, `altaListaBorrar` y `altaListaImprimir`. No es necesario implementar completamente `altaListaImprimir` y `altaListaBorrar`. 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 las funciones `altaListaImprimir` y `altaListaBorrar` necesitan de funciones para saber cómo imprimir y borrar los *datos*, puede utilizar las funciones `estudianteImprimir` y `estudianteBorrar` ya implementadas realizando los *casteos* (conversión de tipos) correspondientes, por ejemplo, de la siguiente manera:

```
altaLista *miAltaLista = altaListaCrear();
altaListaImprimir( miAltaLista, "salida.txt", (tipoFuncionImprimirDato)estudianteImprimir );
altaListaBorrar( miAltaLista, (tipoFuncionBorrarDato)estudianteBorrar );
```

7- **Crear una altaLista nueva, agregarle un estudiante, imprimirla y borrarla.** Ahora sí debe completar las funciones `altaListaImprimir` y `altaListaBorrar`. A esta altura, no es necesario implementar `insertarOrdenado` para agregar un estudiante, puede simplemente utilizar `insertarAtras` que ya está implementada en lenguaje `C` para probar insertando los estudiantes 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:

```
altaLista *miAltaLista = altaListaCrear();
insertarAtras( miAltaLista, estudianteCrear( "leila", "entrania", 21 ) );
altaListaImprimir( miAltaLista, "salida.txt", (tipoFuncionImprimirDato)estudianteImprimir );
altaListaBorrar( miAltaLista, (tipoFuncionBorrarDato)estudianteBorrar );
```

Pruebe insertar varios estudiantes, imprimir la lista y corrobore que los resultados sean los esperados.

8- **Calcular la edad media de los estudiantes.** Deberá implementar `edadMedia`. Pruebe el correcto funcionamiento de la misma, por ejemplo, de la siguiente manera:

```
altaLista *miAltaLista = altaListaCrear();
insertarAtras( miAltaLista, estudianteCrear( "leila", "entrania", 21 ) );
printf( "edadMedia = %2.5f\n", edadMedia( miAltaLista ) );
```

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

9- **Crear una altaLista, agregarle estudiantes 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 *datos*, puede utilizar la función `menorEstudiante` ya implementada realizando el *casteo* (conversión de tipos) correspondiente, por ejemplo, de la siguiente manera:

```
altaLista *miAltaLista = altaListaCrear();
estudiante *miEstudiante = estudianteCrear( "leila", "entrania", 21 ) );
insertarOrdenado( miAltaLista, miEstudiante, (tipoFuncionCompararDato)menorEstudiante );
altaListaImprimir( miAltaLista, "salida.txt", (tipoFuncionImprimirDato)estudianteImprimir );
```

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

10- **Corroborar el invariante de la altaLista.** Si está seguro que conservó y respetó el invariante de la estructura `altaLista` al finalizar cada una de todas las funciones anteriores, puede omitir este punto. Si no, puede aprovechar a chequearlo en este momento. Una forma muy sencilla de hacerlo es con una función que recorra la lista al derecho y luego partiendo del puntero de la `altaLista` al último nodo (`ultimo`), la recorra hacia atrás. Podría aprovechar e ir imprimiendo en consola el resultado de este recorrido. Ya que esta no es una función obligatoria a entregar, y sólo sería una función auxiliar propia para realizar este chequeo de forma manual, podría implementarla rápidamente en lenguaje `C`.

11- **Implementar `filtrarAltaLista`.** Piense detenidamente todos los casos bordes de eliminación de nodos que debe tener en cuenta e impleméntelos. Ya que la función `filtrarAltaLista` necesita de una función para saber cómo comparar *datos*, puede utilizar la función `menorEstudiante` ya implementada realizando el *casteo* (conversión de tipos) correspondiente, por ejemplo, de la siguiente manera:

```
altaLista *miAltaLista = altaListaCrear();
estudiante *Leila = estudianteCrear( "leila", "entrania", 21 ) );
estudiante *Laura = estudianteCrear( "laura", "provoletta", 23 );
insertarOrdenado( miAltaLista, Leila, (tipoFuncionCompararDato)menorEstudiante );
filtrarAltaLista( miAltaLista, (tipoFuncionCompararDato)menorEstudiante, Laura );
```

Pruebe recrear el ejemplo de la Figura 4. Pruebe casos con varios estudiantes de modo que se fuercen los distintos casos bordes implementados en la función. 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:

- `altaLista.h`: Contiene la definición de las estructuras y de las funciones a realizar.
- `altaLista_asm.asm`: Archivo a completar con su código en `lenguaje ensamblador`.
- `lista_c.c`: Archivo con la implementación de las funciones auxiliares 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 `altaLista_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 con el mismo contenido que el dado para realizarlo, habiendo modificado sólo los archivos `altaLista_asm.asm` y `main.c`.

Este TP deberá entregarse como máximo a las 16:59:59 hs del día Jueves 16 de Abril de 2015. ***La reentrega del TP, en caso de ser necesaria, deberá realizarse como máximo a las 16:59:59 hs del día Jueves 7 de Mayo 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.