Trabajo Práctico 1
Super-Lista

Organización del Computador 2

Segundo Cuatrimestre 2012

1. Introducción

El objetivo de este trabajo práctico es implementar un conjunto de funciones sobre una estructura de tipo lista. Las funciones a implementar permiten crear una lista, borrarla, agregar elementos y borrarlos. Además de estas operaciones, se deben proveer dos funciones avanzadas, *filter* y *map*. La primera permite filtrar una lista, quitando elementos que no cumplan una cierta condición y la segunda se encarga de realizar una operación determinada por cada elemento de la lista.

Los ejercicios a realizar para este trabajo práctico estarán divididos en tres secciones. En la primera deberán completar las funciones que permiten manipular el tipo lista. La segunda sección corresponde a un conjunto de funciones muy sencillas que operan sobre tipos de datos básicos. Estas funciones serán utilizadas por el tipo lista para las operaciones avanzadas. En la última sección deben construir un programa en `C` que arme una lista y ejecute algunas de las funciones ejemplo.

1.1. Tipo Lista

La estructura de la lista está compuesta por un bloque de memoria que contiene el puntero al primer nodo de la lista y el tipo de cada uno de estos nodos. Los tipos posibles son `int`, `double` o `string` de C. Por otro lado, el puntero al primer nodo apunta a una estructura de tipo nodo, que contiene dos punteros, uno que indica el siguiente nodo en la lista, y otro que apunta al dato que almacena el nodo. En la figura 1 se puede ver un ejemplo para una lista de strings que contiene 6 palabras.

```
typedef enum tipo_e { sin_tipo=0, tipo_int=1, tipo_double=2, tipo_string=3 } tipo;

typedef struct lista_t {
      enum tipo_e tipo_dato;
      struct nodo_t *primero;
} __attribute__((__packed__)) lista;

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

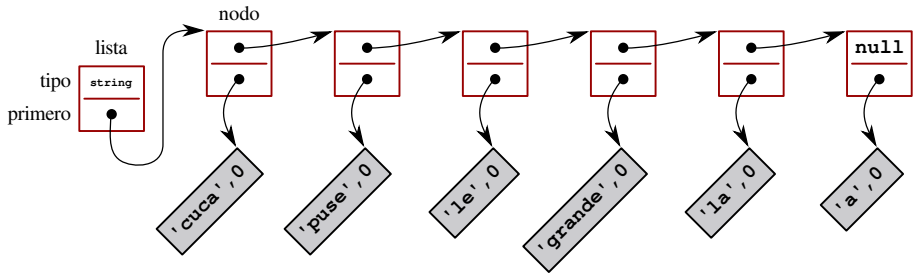
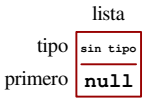


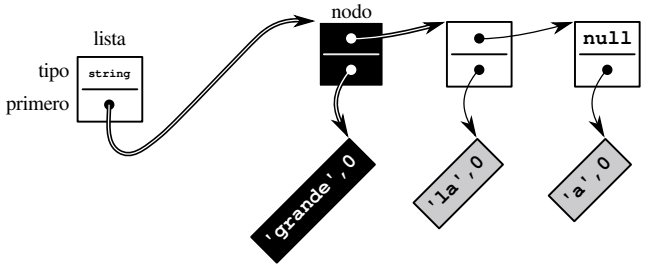
Figura 1: Ejemplo de una estructura lista de `string`s de C. Notar que cada string ocupa una cantidad distinta de bytes.

Funciones de lista

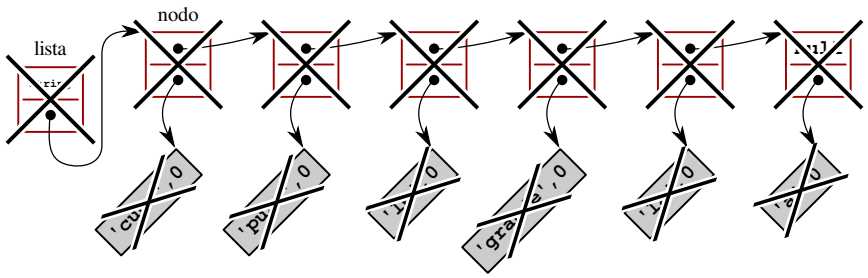
- `lista * listaCrear();`
  Crea una lista vacía, es decir sin ningún nodo y sin indicar el tipo (`sin_tipo`).
  Ej. `lista lt = listaCrear();`



- `void listaInsertar(lista * l, enum tipo_e, void* dato);`
  Inserta un elemento al principio de la lista, siempre que respete el mismo tipo que la lista. En caso contrario no realiza ninguna acción. Si la lista está vacía, le asigna el tipo del elemento a insertar. Al insertar el elemento **debe** hacer una copia del mismo.
  Ej. `dato` es un string de C. `listaInsertar(lt, &dato);`



- `void listaDestruir(lista * l);`
  Borra la lista, eliminando todos los nodos, incluso los datos que contienen.
  Ej. `listaDestruir(lt);`



- `void listaImprimir(lista * l, char *archivo);`
  Debe agregar una línea en el archivo pasado por parámetro que represente a la lista indicada. El archivo se debe abrir en modo *Append*, de modo que las nuevas lineas sean adicionadas al archivo original.
  La lista se debe imprimir en el orden inverso al que los nodos fueron insertados, es decir, desde el último nodo de la lista, al primero. Para imprimir cada nodo, se imprimirá el dato que contenga entre corchetes. Este dato dependerá del tipo, ya sea, un `int`, `double` o una `string` de C.
  Ejemplo para una la lista de la figura 1
  Imprime: `[cuca][puse][le][grande][la][a]`

Funciones avanzadas sobre listas

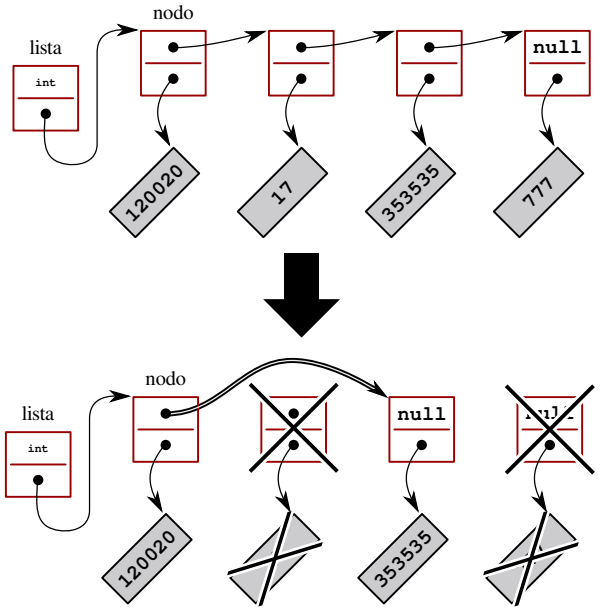
Previo a presentar el comportamiento de las funciones avanzadas cabe mencionar la siguiente definición de tipo.

```
typedef enum boolean_e { false=0, true=1 } boolean;
```

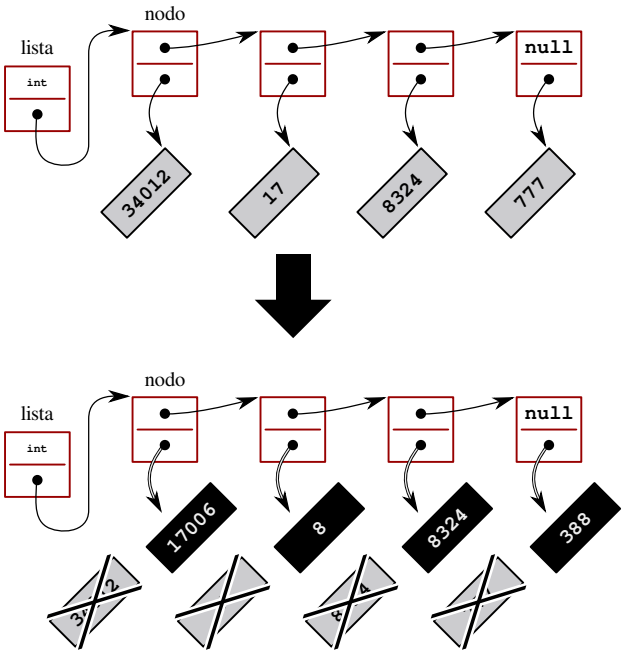
La misma sirve para indicar un valor de verdad.

Las funciones avanzadas a implementar son las siguientes:

- `void listaFilter(lista * l, enum boolean_e (*funcion_filter)(void*) );`
  La función `filter` toma una lista y una función. Evalúa la función para cada elemento de la lista. Si el resultado es "true" entonces conserva el elemento evaluado. En caso contrario borra el elemento y el nodo.
  Ej. `listaFilter(lt, &fun );` donde `fun` es una función que dado un entero retorna si es múltiplo de 5



- `void listaMap(lista * l, void* (*funcion_map)(void*) );`
  La función `map` toma una lista y una función. Evalúa la función para cada elemento de la lista, creando un **nuevo** elemento que debe ser asignado a la posición del elemento previamente evaluado. Notar que el elemento que fue evaluado por la función **debe** ser eliminado.
  Ej. `listaMap(lt, &fun );` donde `fun` es una función que dado un puntero a un entero, retorna un nuevo puntero a un entero, con el valor dividido por 2.



Funciones adicionales

Las funciones avanzadas requieren como parámetros otras funciones. A continuación se enumeran las únicas 6 funciones que serán utilizadas como parámetros. Tres de ellas se usarán para la función `filter`, y las tres restantes para la función `map`.

- `enum boolean_e es_multiplo_de_5(int* dato);`
  Toma un puntero a un *int* y retorna un valor de verdad que indica si el número es múltiplo de 5.

- `enum boolean_e es_negativo(double* dato);`
  Toma un puntero a un *double* y retorna un valor de verdad que indica si el número es negativo.

- `enum boolean_e es_largo_mayor_10(char* dato);`
  Toma una *string* de C y retorna un valor de verdad que indica si la longitud de la cadena de caracteres es mayor o igual a 10.

- `int* dividir_por_dos(int* dato);`
  Toma un puntero a un *int* y retorna un **nuevo** puntero a un *int* que contiene el valor dividido por 2.

- `double* multiplicar_por_pi(double* dato);`
  Toma un puntero a un *double* y retorna un **nuevo** puntero a un *double* que contiene el valor multiplicado por π.

- `char* tomar_primeros_10(char* dato);`
  Toma una *string* de C y retorna una **nueva** *string* de C que contiene una copia de los primeros k caracteres, con $k = min(10, logintud)$.

2. Enunciado

Ejercicio 1

Deberá implementar una serie de funciones en assembler que se enumeran a continuación.

- Funciones de lista
  - `lista * listaCrear();`
  - `void listaInsertar(lista * l, enum tipo_e, void* dato);`
  - `void listaDestruir(lista * l);`
  - `void listaImprimir(lista *l, char *archivo);`
- Funciones avanzadas de lista
  - `void listaFilter(lista * l, void* (*funcion_filter)(void*) );`
  - `void listaMap(lista * l, void* (*funcion_map)(void*) );`
- Funciones auxiliares para `listaFilter`
  - `enum boolean_e es_multiplo_de_5(int* dato);`
  - `enum boolean_e es_negativo(double* dato);`
  - `enum boolean_e es_largo_mayor_10(char* dato);`
- Funciones auxiliares para `listaMap`
  - `int* dividir_por_dos(int* dato);`
  - `double* multiplicar_por_pi(double* dato);`
  - `char* tomar_primeros_10(char* dato);`

Ejercicio 2

Construir un programa de prueba (`main.c`) que realice las siguientes acciones llamando a las funciones implementadas anteriormente:

1- Crear una lista nueva (`lista lt = listaCrear()`).

2- Insertar los siguientes 4 elementos como *int*: `325`, `823`, `100` y `105`.

3- Aplicar la función `listaFilter(lt, &es_multiplo_de_5 )`, pasando como parámetro la función que filtra múltiplos de 5.

4- Aplicar la función `listaMap(lt, &dividir_por_dos )`, pasando como parámetro la función que divide por 2.

5- Imprimir la lista.

6- Destruir la lista .

Realizar el mismo procedimiento para:

1) Una lista de *double*: `-3.25`, `8.23`, `1.00` y `-1.05`, aplicando las funciones `listaFilter(lt, &es_negativo )` y `listaMap(lt, &multiplicar_por_pi )`.

2) Una lista de *strings* de C: `'Ricardo, Ricardo, Ricardo Ruben'`, `'mono'` y `'en escalada no hay nada'`, aplicando las funciones `listaFilter(lt, &es_largo_mayor_10 )` y `listaMap(lt, &tomar_primeros_10 )`.

Ejercicio 3 - Optativo

- Proponga un orden alternativo para las palabras de la lista del ejemplo (fig. 1).

Testing

En un ataque de bondad, hemos decidido entregarle una serie de *tests* o pruebas para que usted mismo pueda verificar el buen funcionamiento de su código.

Luego de compilar, puede ejecutar `./test.sh` y eso probará su código. Un test consiste en la creación, inserción, eliminación e impresión en archivo de varios elementos de una lista según corresponda. 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.

Archivos

Se entregan los siguientes archivos:

- `tp.c`: Es el archivo principal, contiene el método `main`. No debe modificarlo.
- `lista.h`: Contiene la definición de la estructura de la lista, los elementos y las funciones que debe completar.
- `lista.asm`: Archivo a completar con su código.
- `Makefile`: Contiene las instrucciones para compilar el código.
- `test.sh`: Es el script que realiza todos los test.

Notas:

a) Todas las funciones deben estar en lenguaje ensamblador. Cualquier función extra, también debe estar hecha en lenguaje ensamblador.

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

c) Para el manejo de archivos se recomienda usar las funciones de C: `fopen`, `fread`, `fwrite`, `fclose`, `fseek`, `ftell`, `fprintf`, etc.

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. El criterio de aprobación es que el TP supere correctamente todos los tests.

3. Informe y forma de entrega

Para este trabajo práctico no deberán entregar un informe. En cambio, deberán entregar un archivo comprimido con el mismo contenido que el dado para realizarlo, habiendo modificado solo archivo `lista.asm`, y agregado un archivo `main.c` con el segundo ejercicio resuelto. Además se deberá adjuntar un *Makefile* para compilar este último.

La fecha de entrega de este trabajo es Jueves 13 de Septiembre. Deberá ser entregado a través de la página web. El sistema sólo aceptará entregas de trabajos hasta las 17:00 hs del día de entrega.

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