Trabajo Práctico 1
Popurrí de estructuras: Enteros, Strings, Listas y Matrices

Organización del Computador II

Segundo Cuatrimestre 2018

1. Introducción

El objetivo de este TP es implementar un conjunto de funciones sobre distintas estructuras para almacenar datos. Las estructuras a implementar serán, enteros, listas, strings y matrices. Estos contenedores serán utilizados de forma recursiva, es decir, es posible tener una matriz que contenga listas de enteros y strings. A su vez, cada contenedor tendrá una función para borrarse y otra para imprimir. Estas funciones serán llamadas recursivamente desde otros contenedores.

2. Estructuras

Cada estructura será identificada por un *id* que respeta el siguiente enumerado:

```
typedef enum e_type {
    NONE    = 0,
    INTEGER = 1,
    STRING  = 2,
    LIST    = 3,
    MATRIX  = 4,
} type_t;
```

Las aridades de los punteros a función almacenados en las estructuras o pasados por parámetro son los 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 Enteros

Almacena un solo valor entero con signo de 32 bits.

```
typedef struct s_integer{
    type_t dataType;
    funcDelete_t *remove;
    funcPrint_t *print;
    int *data;
} integer_t;
```

Estructura String

Almacena un puntero a una string de C terminada en cero.

```
typedef struct s_string{
    type_t dataType;
    funcDelete_t *remove;
    funcPrint_t *print;
    char *data;
} string_t;
```

Estructura Lista

Implementa una lista simplemente encadenada de nodos. La estructura `lista_t` contiene un puntero al primer elemento de la lista, mientras que cada elemento nodo de tipo `listElem_t` contiene un puntero al siguiente y otro al dato almacenado.

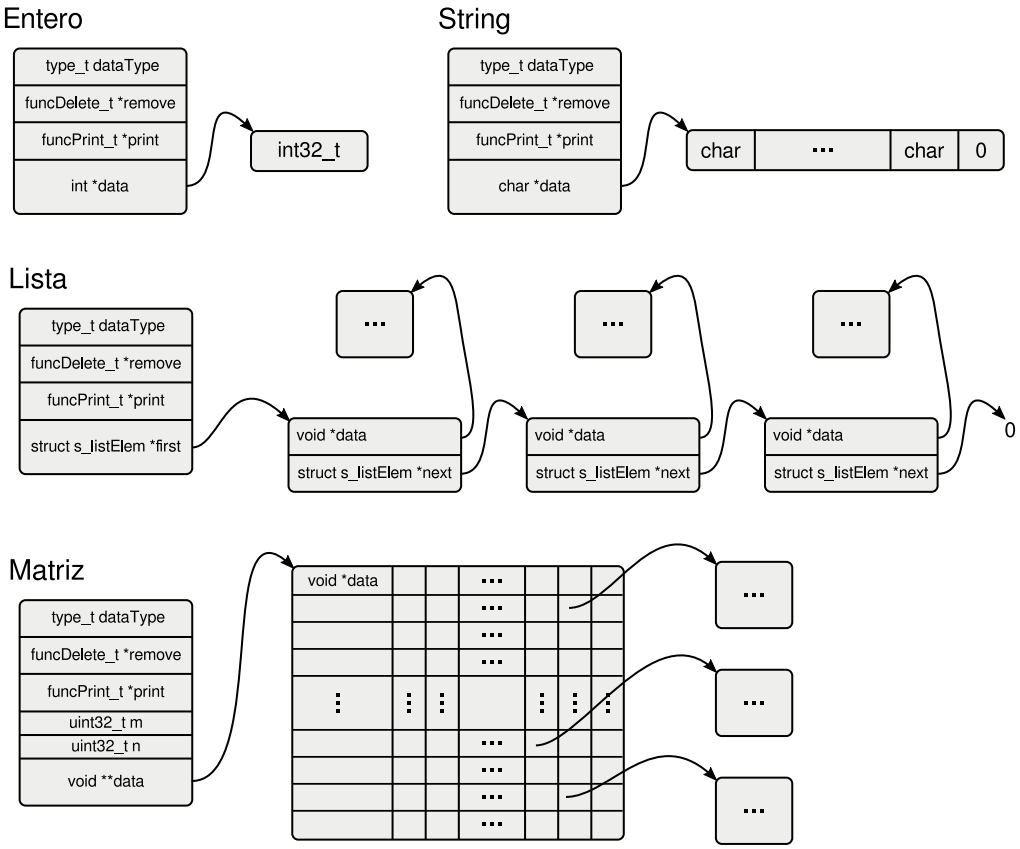
```
typedef struct s_listElem{
    struct s_element *data;
    struct s_listElem *next;
} listElem_t;

typedef struct s_list{
    type_t dataType;
    funcDelete_t *remove;
    funcPrint_t *print;
    struct s_listElem *first;
} list_t;
```

Estructura Matriz

Implementa una matriz de datos. La estructura `matrix_t` almacena un arreglo de punteros a datos de tamaño `m`×`n`, donde `m` es la cantidad de columnas y `n` la cantidad de filas.

```
typedef struct s_matrix{
    type_t dataType;
    funcDelete_t *remove;
    funcPrint_t *print;
    uint32_t m;
    uint32_t n;
    void **data;
} matrix_t;
```

Ejemplos



3. Funciones

En está sección se describen las diferentes funciones para operar con las estructuras.

Funciones de Enteros

- `integer_t* intNew();`
  Crea una nueva estructura `integer_t` vacía.
- `integer_t* intSet(integer_t* i, int d);`
  Copia un nuevo valor a una estructura `integer_t`. La misma puede tener o no un valor previamente cargado.
- `integer_t* intRemove(integer_t* i);`
  Borra un dato cargado en la estructura, pero no borra la estructura.
- `void intDelete(integer_t* i);`
  Borra toda la estructura con su dato, esté cargado o no.
- `int32_t intCmp(integer_t* a, integer_t* b);`
  Compara los valores enteros cargados en las estructuras. Si `a` y `b` son iguales, retorna cero. Si `a<b` retorna 1 y si `a>b` retorna -1.
- `void intPrint(integer_t* i, FILE *pFile);`
  Escribe en `pFile` el entero almacenado en `i`, de no existir escribe `NULL`.

Funciones de String

- `string_t* strNew();`
  Crea una nueva estructura `string_t` vacía.
- `string_t* strSet(string_t* s, char* c);`
  Copia la string `c` a la estructura `s`. La misma ya puede o no, tener una string cargada previamente.
- `string_t* strAddRight(string_t* s, string_t* d);`
  Carga en la string `s` la concatenación de strings `s+d`, borrando la string `d`.
- `string_t* strAddLeft(string_t* s, string_t* d);`
  Carga en la string `s` la concatenación de strings `d+s`, borrando la string `d`.
- `string_t* strRemove(string_t* s);`
  Borra la string cargada en la estructura, pero no borra la estructura.
- `void strDelete(string_t* s);`
  Borra toda la estructura incluyendo el string, esté cargado o no.
- `int32_t strCmp(string_t* a, string_t* b);`
  Compara dos strings en orden lexicográfico. Si `a` y `b` son iguales, retorna cero. Si `a<b` retorna 1 y si `a>b` retorna -1.
- `void strPrint(string_t* s, FILE *pFile);`
  Escribe en `pFile` la string almacenada en `s`, de no existir escribe `NULL`.

Funciones de Lista

- `list_t* listNew();`
  Crea una nueva estructura `list_t` vacía.
- `list_t* listAddFirst(list_t* l, void*  data);`
  Agrega un nuevo nodo al principio de la lista.
- `list_t* listAddLast(list_t* l, void* data);`
  Agrega un nuevo nodo al final de la lista.

- `list_t* listAdd(list_t* l, void* data, funcCmp_t* f);`
  Agrega un nuevo nodo a la lista, respetando la función de orden `f`.
- `list_t* listRemoveFirst(list_t* l);`
  Borra el primer nodo de la lista.
- `list_t* listRemoveLast(list_t* l);`
  Borra el último nodo de la lista.
- `list_t* listRemove(list_t* l, void* data, funcCmp_t* f);`
  Borra todos los nodos de la lista tal que la función de comparación indique igualdad.
- `void listDelete(list_t* l);`
  Borra la lista. Recursivamente borra todos los datos que contiene.
- `void listPrint(list_t* l, FILE *pFile);`
  Escribe en `pFile` la lista almacenada en `l`. Para cada elemento llama recursivamente a la función imprimir del elemento almacenado. El formato de la lista será: `[`x_0`,...,`x_{n-1}`]`, suponiendo x_i como el resultando de llamar a la función imprimir del elemento almacenado.

Funciones de Matriz

- `matrix_t* matrixNew(uint32_t m, uint32_t n);`
  Crea una nueva estructura `matrix_t` vacía. Considerar que puede o no crear la matriz donde almacenar la información.
- `matrix_t* matrixAdd(matrix_t* m, uint32_t x, uint32_t y, void* data);`
  Agrega un dato a la matriz en la columna `x`, fila `y`.
- `matrix_t* matrixRemove(matrix_t* m, uint32_t x, uint32_t y);`
  Borra el dato almacenado en la columna `x`, fila `y`.
- `void matrixDelete(matrix_t* m);`
  Borra la matriz. Recursivamente borra todos los datos que contiene.
- `void matrixPrint(matrix_t* m, FILE *pFile);`
  Escribe en `pFile` la matriz almacenada en `m`. Para cada elemento llama recursivamente a la función imprimir del elemento almacenado. El formato de la matriz para cada fila será: `|`x_0`|...|`x_{n-1}`|`, suponiendo x_i como el resultando de llamar a la función imprimir del elemento almacenado. Al final de cada fila se agregará un salto de línea.

Funciones auxiliares recomendadas

- `uint32_t str_len(char* a)`
  Devuelve la cantidad de caracteres de la string.
- `char* str_copy(char* a)`
  Devuelve una nueva copia de una string terminada en cero.
- `int32_t str_cmp(char* a, char* b)`
  Compara dos *strings* terminadas en 0, en orden lexicográfico. Debe retornar:
  - 0 si son iguales
  - 1 si `a` $<$ `b`
  - -1 si `b` $<$ `a`
- `char* str_concat(char* a, char* b)`
  Genera una nueva string con la concatenación de `a` y `b`.

Consideraciones

- La estructura `integer_t` y `string_t` puede tener su puntero a datos en cero.
- Para la función `listAdd` y `listRemove` todos los elementos de la lista deben tener el tipo de la función que se utiliza para comparar.
- Los datos se agregan a estructuras `integer_t` o `string_t` por copia. Generan una copia del dato pasado por parámetro.
- Los datos se agregan a estructuras `list_t` o `matrix_t` por referencia. Se almacena el puntero al dato pasado por parámetro.

4. Enunciado

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

- `integer_t* intNew();`(11 Inst.)
- `integer_t* intSet(integer_t* i, int d);`(15 Inst.)
- `integer_t* intRemove(integer_t* i);`(11 Inst.)
- `void intDelete(integer_t* i);`(6 Inst.)
- `int32_t intCmp(integer_t* a, integer_t* b);`(10 Inst.)
- `void intPrint(integer_t* m, FILE *pFile);`(14 Inst.)
- `string_t* strNew();`(12 Inst.)
- `string_t* strSet(string_t* s, char* c);`(14 Inst.)
- `string_t* strAddRight(string_t* s, string_t* d);`(21 Inst.)
- `string_t* strAddLeft(string_t* s, string_t* d);`(21 Inst.)
- `string_t* strRemove(string_t* s);`(11 Inst.)
- `void strDelete(string_t* s);`(6 Inst.)
- `int32_t strCmp(string_t* a, string_t* b);`(3 Inst.)
- `void strPrint(string_t* m, FILE *pFile);`(13 Inst.)
- `list_t* listNew();`(12 Inst.)
- `list_t* listAddFirst(list_t* l, void*  data);`(16 Inst.)
- `list_t* listAddLast(list_t* l, void* data);`(24 Inst.)
- `list_t* listAdd(list_t* l, void* data, funcCmp_t* f);`(43 Inst.)
- `list_t* listRemove(list_t* l, void* data, funcCmp_t* f);`(36 Inst.)
- `list_t* listRemoveFirst(list_t* l);`(19 Inst.)
- `list_t* listRemoveLast(list_t* l);`(25 Inst.)
- `void listDelete(list_t* l);`(20 Inst.)
- `void listPrint(list_t* m, FILE *pFile);`(35 Inst.)
- `matrix_t* matrixAdd(matrix_t* m, uint32_t x, uint32_t y, void* data);`(22 Inst.)
- `matrix_t* matrixRemove(matrix_t* m, uint32_t x, uint32_t y);`(23 Inst.)
- `void matrixDelete(matrix_t* m);`(26 Inst.)

Se recomienda además implementar las siguientes funciones en lenguaje ensamblador.

- `uint32_t str_len(char* a)`(7 Inst.)
- `char* str_copy(char* a)`(19 Inst.)
- `int32_t str_cmp(char* a, char* b)`(25 Inst.)
- `char* str_concat(char* a, char* b)`(36 Inst.)

Además las siguientes funciones pueden ser implementadas en lenguaje C.

- `matrix_t* matrixNew(uint32_t m, uint32_t n);`(35 Inst.)
- `void matrixPrint(matrix_t* m, FILE *pFile);`(64 Inst.)

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 `./pruebacorta.sh`. Para compilar el código y correr las pruebas intensivas deberá ejecutar `./prueba.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 Lista** Construir una lista vacía. Agregar en orden los siguientes números: 34, 42, 13, 44, 58, 11, 92. Imprimir la lista en un archivo. Borrar el número 13 y el 58. Imprimir nuevamente la lista en le mismo archivo.

2- **Caso Matriz** Construir la siguiente matriz, imprimirla en un archivo y borrar el dato.

```
|NULL|[1,2,3]|[SA,RA,SA]|NULL|
|NULL|[[],2,3]|[SA,RA,SA]|NULL|
|NULL|[[],[],3]|[[SA],[RA],[SA]]|35|
|NULL|[[SA],[RA],[SA]]|[1,[2],3]|32|
|NULL|[[ra,ra,33],[ro,ro,35]]|8|31|
```

El programa puede correrse con `./pruebacorta.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 `./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.

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` y `fclose`.
- Para el manejo de strings **no está permitido** usar las funciones de `C`: `strlen`, `strcpy`, `strcmp` y `strdup`.
- Para correr los test requiere *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:

- `matriz_lista_string.h`: Contiene la definición de las estructuras y de las funciones a realizar. No debe ser modificado.
- `matriz_lista_string.asm`: Archivo a completar con la implementación de las funciones en lenguaje ensamblador.
- `matriz_lista_string.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 (`pruebacorta.sh`).
- `tester.c`: Archivo del tester de la cátedra. No debe ser modificado.
- `pruebacorta.sh`: Script para ejecutar el test simple (pruebas cortas). No debe ser modificado.
- `prueba.sh`: Script para ejecutar todos los test intensivos. No debe ser modificado.
- `salida.caso*.catedra.txt`: Archivos de salida para comparar con sus salidas. No debe ser modificado.

5. 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 `matriz_lista_string.asm`, `matriz_lista_string.c` y `main.c`.

La fecha de entrega de este trabajo es 6/09. 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=1370&section=3`). Ante cualquier problema con la entrega, comunicarse por mail a la lista de docentes `orga2-doc@dc.uba.ar`.