Trabajo Práctico 1
Criptomentiritas
(Adenda)

Organización del Computador II

Primer Cuatrimestre 2018

1. Introducción

El objetivo de este TP es implementar un conjunto de funciones sobre una estructura recursiva que representa una lista de funciones que se aplican de forma progresiva sobre strings.

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 operaciones 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.

En este trabajo en particular el nodo guarda referencia a dos funciones y a un tipo de operación. La lista representa una serie de operaciones a realizar sobre un string dado, cada nodo modificará el string haciendo uso de una de las operaciones. Las operaciones pueden ser reversibles o irreversibles. Si resultan reversibles dentro del nodo las operaciones serán complementarias, si $f_i : \Sigma^* \to \Sigma^*$ y $g_i : \Sigma^* \to \Sigma^*$ son las operaciones sobre strings en el nodo s_i luego $f \circ g = id$. Si no resultan reversibles la operación aplicada sobre el string puede llegar a destruir la información.

Vamos a definir una operación sobre la lista llamada $apply : \Sigma^* \to \Sigma^*$ que aplica sucesivamente una de las funciones a un string. Por lo general vamos a querer codificar el string en una dirección:

$$encode(s) : f_k(\dots f_2(f_1(s)) \dots)$$

Y si queremos decodificar un string al que aplicamos operaciones revesibles haremos:

$$decode(s) : g_1(\dots g_{k-1}(g_k(s)) \dots)$$

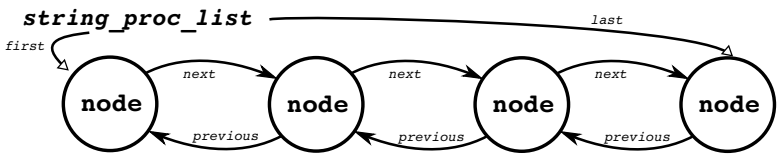

Figura 1: Ejemplo de Lista Doblemente Enlazada.

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 string_proc_list, string_proc_node y string_proc_node

La lista `string_proc_list` es la estructura macro que guarda referencia al primer y último elemento que son de tipo `string_proc_node`. Tiene un atributo de nombre que permite discriminar las distintas listas en aquellos contextos en los que se use más de una.
Los nodos `string_proc_node` a su vez guardan una referencia al próximo (`next`) y anterior (`previos`) elemento de la lista para permitir implementar una lista doblemente enlazada, una referencia a las funciones de codificación (`f`)y decodificación (`g`) que están definidas como punteros a función `void (*string_proc_func) (string_proc_key*)`.

```
typedef struct string_proc_list_t {
        char* name;
        struct string_proc_node_t* first;
        struct string_proc_node_t* last;
} __attribute__((__packed__)) string_proc_list;

typedef struct string_proc_node_t {
        struct string_proc_node_t* next;
        struct string_proc_node_t* previous;
        string_proc_func f;
        string_proc_func g;
        string_proc_func_type type;
} __attribute__((__packed__)) string_proc_node;

typedef void (*string_proc_func) (string_proc_key*);
```

El tipo `type` hace referencia a un atributo `enum` que permite indicar si las funciones que el nodo ha de aplicar son reversibles (`REVERSIBLE`) o no (`IRREVERSIBLE`).

```
typedef enum string_proc_func_type_t{
        REVERSIBLE                = 0,
        IRREVERSIBLE        = 1
} __attribute__((__packed__)) string_proc_func_type;
```

Las operaciones se aplican sobre un tipo `string_proc_key` que contiene un valor `value` que es el contenido efectivo sobre el que queremos operar y el largo `length`.

```
typedef void (*string_proc_func) (string_proc_key*);
typedef struct string_proc_key_t {
        uint32_t length;
        char* value;
} __attribute__((__packed__)) string_proc_key;
```

1.2. Definiciones de punteros a funciones para codificar, decodificar un string

Vamos definir una serie de funciones sobre `string_proc_key` que transformen su contenido. Vamos a aplicar las funciones sobre el atributo valor pero teniendo cuidado de mantener el caracter de cierre intacto, ya que es el que podríamos estar utilizando para imprimir o determinar el largo efectivo del string. Estas serán las funciones cuyas referencias contengan los nodos en los atributos f y g.
Las funciones que vamos a querer aplicar sobre nuestras cadenas son las siguientes:

- `void shift_2(string_proc_key* key)`
  Que desplaza cada caracter dos posiciones hacia arriba en su codificación `ASCII` siguiendo una lógica de wrap-around, esto quiere decir que se aplica como una congruencia módulo 256. Este par de funciones son reversibles.

- `void unshift_2(string_proc_key* key)`
  Que desplaza cada caracter dos posiciones hacia abajo en su codificación `ASCII` siguiendo una lógica de wrap-around, esto quiere decir que se aplica como una congruencia módulo 256. Este par de funciones son reversibles.

- `void shift_position(string_proc_key* key)`
  Que desplaza cada caracter tantas posiciones hacia arriba en su codificación `ASCII` como sea su posición dentro de la cadena, por ejemplo si un caracter c se encuentra en la posición 3 de la cadena

se lo desplazara 3 posiciones hacia arriba, siguiendo una lógica de wrap-around, esto quiere decir que se aplica como una congruencia módulo 256. Este par de funciones son reversibles.

- `void unshift_position(string_proc_key* key)`
  Que desplaza cada caracter tantas posiciones hacia abajo en su codificación `ASCII` como sea su posición dentro de la cadena, por ejemplo si un caracter c se encuentra en la posición 3 de la cadena se lo desplazara 3 posiciones hacia abajo, siguiendo una lógica de wrap-around, esto quiere decir que se aplica como una congruencia módulo 256. Este par de funciones son reversibles.

- `void saturate_2(string_proc_key* key)`
  Que desplaza cada caracter dos posiciones hacia arriba en su codificación `ASCII` siguiendo una lógica de saturación, o sea, si el valor supera a 255 debe mantenerse en 255. Este par de funciones no son reversibles.

- `void unsaturate_2(string_proc_key* key)`
  Que desplaza cada caracter dos posiciones hacia abajo en su codificación `ASCII` siguiendo una lógica de saturación, o sea, si el valor es menor a 0 debe mantenerse en 0.Este par de funciones no son reversibles.

- `void saturate_position(string_proc_key* key)`
  Que desplaza cada caracter tantas posiciones hacia arriba en su codificación `ASCII` como sea su posición dentro de la cadena siguiendo una lógica de saturación, o sea, si el valor supera a 255 debe mantenerse en 255. Este par de funciones no son reversibles.

- `void unsaturate_position(string_proc_key* key)`
  Que desplaza cada caracter tantas posiciones hacia abajo en su codificación `ASCII` como sea su posición dentro de la cadena siguiendo una lógica de wrap-around, esto quiere decir que se aplica como una congruencia módulo 256. Este par de funciones no son reversibles.

- `void saturate_2_odd(string_proc_key* key)`
  Que desplaza cada caracter dos posiciones hacia arriba en su codificación `ASCII` siguiendo una lógica de saturación, o sea, si el valor es menor a 0 debe mantenerse en 0, pero sólo si la posición del caracter es impar, caso contrario deja el caracter invariante. Este par de funciones no son reversibles.

- `void unsaturate_2_odd(string_proc_key* key)`
  Que desplaza cada caracter dos posiciones hacia abajo en su codificación `ASCII` siguiendo una lógica de saturación, o sea, si el valor es menor a 0 debe mantenerse en 0,, pero sólo si la posición del caracter es impar, caso contrario deja el caracter invariante. Este par de funciones no son reversibles.

- `void shift_position_prime(string_proc_key* key)`
  Que desplaza cada caracter tantas posiciones hacia arriba en su codificación `ASCII` como sea su posición dentro de la cadena, por ejemplo si un caracter c se encuentra en la posición 3 de la cadena se lo desplazara 3 posiciones hacia arriba, pero sólo si la posición del caracter es prime, caso contrario deja el caracter invariante, siguiendo una lógica de wrap-around, esto quiere decir que se aplica como una congruencia módulo 256. Este par de funciones son reversibles.

- `void unshift_position_prime(string_proc_key* key)`
  Que desplaza cada caracter tantas posiciones hacia abajo en su codificación `ASCII` como sea su posición dentro de la cadena, por ejemplo si un caracter c se encuentra en la posición 3 de la cadena se lo desplazara 3 posiciones hacia arriba, pero sólo si la posición del caracter es prime, caso contrario deja el caracter invariante, siguiendo una lógica de wrap-around, esto quiere decir que se aplica como una congruencia módulo 256. Este par de funciones son reversibles.

1.3. Funciones de string_proc_list

- `string_proc_list* string_proc_list_create(char* name);`
  Pide memoria para una estructura de lista, copia el contenido de `name` en su atributo correspondiente e inicializa los punteros `first` y `last` en `NULL`.

- `string_proc_node* string_proc_node_create(string_proc_func f, string_proc_func g, string_proc_func_type type);`
  Pide memoria para una estructura de nodo y asigna los valores pasados por parámetros a los atributos correspondientes.

- `string_proc_key* string_proc_key_create(char* value);`
  Pide memoria para una estructura de nodo y asigna el atributo de `length` de acuerdo al largo de `value` y copia el valor pasado por parámetro en su atributo correspondiente.

- `void string_proc_list_destroy(string_proc_list* list);`
  Libera las estructuras anidadas, haciendo llamadas a `string_proc_node_destroy` de ser necesario y devuelve los atributos a su estado inicial.

- `void string_proc_node_destroy(string_proc_node* node);`
  Libera la memoria que haya reservado y devuelve los atributos a su estado inicial.

- `void string_proc_key_destroy(string_proc_key* key);`
  Libera la memoria que haya reservado y devuelve los atributos a su estado inicial.

- `void string_proc_list_add_node(string_proc_list* list, string_proc_func f, string_proc_func g, string_proc_func_type type);`
  Crea una estructura de nodo con los parámetros `f`,`g` y `type`, actualiza el estado de la lista, tanto en la estructura contenedora como en los nodos pre-existentes.

- `void string_proc_list_apply(string_proc_list* list, string_proc_key* key, bool encode);`
  Toma la clave `key` pasada por parámetro y le aplica las funciones contenidas en los nodos de la lista, si el valor `encode` es `true` irá aplicando la función `f` de cada nodo empezando por el primero hasta llegar al último, caso contrarion aplicará la función `g` de cada nodo empezando del último hasta llegar al primero.

- `void string_proc_list_print(string_proc_list* list, FILE* file);`
  Esta función imprime el contenido de la lista en la salida especificada en `file`, el formato debe ser el siguiente:

  ```
  List length:2
          node type:reversible
          node type:reversible
  ```

  Comenzando con la cadena `List length:` luego con la cantidad de elementos en la lista y luego una línea por cada nodo con una tabulación de comienzo, luego la cadena `node type:` y la palabra `reversible` o `irreversible` según corresponda.

- `void string_proc_list_encode(string_proc_list* list, char* msg, FILE* file);`
  Esta función codifica una cadena pasada por parámetro en `msg`, para esto debe crear la estructura `string_proc_key` correspondiente y hacer la llamada a la función `string_proc_list_apply` con los valores adecuados, para permitir aplicar las operaciones `f` desde el comienzo de la lista hacia el final. El resultado final, contenido en el atributo `value` de la estructura `string_proc_key` debe imprimirse en la salida `file`.

- `void string_proc_list_decode(string_proc_list* list, char* msg, FILE* file);`
  Esta función codifica una cadena pasada por parámetro en `msg`, para esto debe crear la estructura `string_proc_key` correspondiente y hacer la llamada a la función `string_proc_list_apply` con los valores adecuados, para permitir aplicar las operaciones `g` desde el final de la lista hacia el comienzo. El resultado final, contenido en el atributo `value` de la estructura `string_proc_key` debe imprimirse en la salida `file`.

- `uint32_t string_proc_list_type_amount(string_proc_list* list, string_proc_func_type type);`
  Esta función devuelve la cantiad de nodos dentro de la lista cuyo tipo se corresponde con el pasado por parámetro `type`.

- `string_proc_list* string_proc_list_filter_by_type(string_proc_list* list, string_proc_func_type type);`
  Esta función devuelve una lista nueva en base a la lista `list` copiando en ella los nodos cuyo tipo se corresponde con el pasado por parámetro `type`.

- `string_proc_list* string_proc_list_invert(string_proc_list* list);`
  Esta función devuelve una nueva lista, cuyos nodos son equivalentes a los nodos de la lista original `list` pero con la salvedad de que sus funciones `f` y `g` han sido intercambiadas.

- `uint32_t string_proc_list_length(string_proc_list* list);`
  Esta función devuelve el largo de la lista pasada por parámetro.

- `bool string_proc_list_add_node_at(string_proc_list* list, string_proc_func f, string_proc_func g, string_proc_func_type type, uint32_t index);`
  Debe insertar el nodo con los parámetros correspondientes en la posición indicada por index desplazando en una posición hacia adelante los nodos sucesivos en caso de ser necesario, la estructura de la lista debe ser actualizada de forma acorde. Si index es igual al largo de la lista debe insertarlo al final de la misma, si index es mayor al largo de la lista no debe insertar el nodo. Debe devolver true si el nodo pudo ser insertado en la lista, false en caso contrario

- `bool string_proc_list_remove_node_at(string_proc_list* list, uint32_t index);`
  Debe eliminar el nodo que se encuentra en la posición indicada por index de ser posible la lista debe ser actualizada de forma acorde y debe devolver true si pudo eliminar el nodo o false en caso contrario.

- `string_proc_list* string_proc_list_invert_order(string_proc_list* list);`
  Debe devolver una copia de la lista pasada por parámetro copiando los nodos en el orden inverso.

- `void string_proc_list_apply_print_trace(string_proc_list* list, string_proc_key* key, bool encode, FILE* file);`
  Hace una llamada sucesiva a los nodos de la lista pasada por parámetro siguiendo la misma lógica que `string_proc_list_apply` pero comienza imprimiendo una línea `Encoding key 'valor_de_la_clave'through list nombre_de_la_lista\n` y luego por cada aplicación de una función f o g escribe `Applying function at [direccion_de_funcion] to get 'valor_de_la_clave'\n`, un ejemplo sería:

  ```
  Decoding key 'hemos ido demasiado...' through list lista vacía
  Applying function at [0x10b9c0250] to get 'fckmqgbmbck_...'
  Applying function at [0x10b9c0090] to get 'daikoe‘k‘ai]oe]...'
  Applying function at [0x10b9c0250] to get 'b_gimc^i^_g[m...'
  Applying function at [0x10b9c0330] to get 'b^efi]WaTT[N_...'
  ```

1.4. Funciones Auxiliares Sugeridas

- `uint32_t str_len(char* a);`
  Devuelve la cantidad de caracteres de `s`.

- `char* str_copy(char* a);`
  Devuelve una nueva copia de `s`.

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

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

- `int32_t str_cmp(char* a, char* b);`
  Indica si `a` $<$ `b`, 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**.
  De este modo si a es menor que b devuelve 1, si b es menor que a devuelve -1 y si son iguales devuelve 0.

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:

- `string_proc_list* string_proc_list_create(char* name);`
- `string_proc_node* string_proc_node_create(string_proc_func f, string_proc_func g, string_proc_func_type type);`
- `string_proc_key* string_proc_key_create(char* value);`
- `void string_proc_list_destroy(string_proc_list* list);`
- `void string_proc_node_destroy(string_proc_node* node);`
- `void string_proc_key_destroy(string_proc_key* key);`
- `void string_proc_list_add_node(string_proc_list* list, string_proc_func f, string_proc_func g, string_proc_func_type type);`
- `void string_proc_list_apply(string_proc_list* list, string_proc_key* key, bool encode);`

Las funciones a implementar en C son

- `uint32_t string_proc_list_length(string_proc_list* list);`
- `bool string_proc_list_add_node_at(string_proc_list* list, string_proc_func f, string_proc_func g, string_proc_func_type type, uint32_t index);`
- `bool string_proc_list_remove_node_at(string_proc_list* list, uint32_t index);`
- `string_proc_list* string_proc_list_invert_order(string_proc_list* list);`
- `void string_proc_list_apply_print_trace(string_proc_list* list, string_proc_key* key, bool encode, FILE* file);`
- `void saturate_2_odd(string_proc_key* key)`
- `void unsaturate_2_odd(string_proc_key* key)`
- `void shift_position_prime(string_proc_key* key)`
- `void unshift_position_prime(string_proc_key* key)`

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- En los siguientes casos imprima los estados de la lista: cree una lista vacía, agregue y quite un elemento al comienzo. Cree una lista con cinco nodos irreversibles y: agregue y quite al comienzo un nodo reversible, agregue y quite al final un nodo reversible, intente agregar y quitar un nodo reversible fuera de rango, agregue y quite un nodo reversible en la posición 2.

2- Cree una lista con cinco dos nodos irreversibles, uno reversible y uno irreversible (en ese orden), imprímala, consiga la copia de orden inverso (`string_proc_list_invert_order`) e imprima la copia.

3- Cree una lista con cinco dos nodos irreversibles, uno reversible y uno irreversible (en ese orden), imprímala y haga la llamada a `string_proc_list_apply_print_trace` sobre la clave de valor `"hemos ido demasiado lejos y se acerca la hora de detenernos a reflexionar"`.

4- Purebe las funciones `saturate_2_odd, unsaturate_2_odd, shift_position_prime, unshift_position_prime` sobre el string `hemos ido demasiado lejos y se acerca la hora de detenernos a reflexionar`

2.2.2. 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.

2.3. Archivos

Se entregan los siguientes archivos:

- `string_processor.h`: Contiene la definición de las estructuras y de las funciones a realizar.
- `string_processor_asm.asm`: Archivo a completar con su código en `lenguaje ensamblador`.
- `string_processor_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 `string_processor_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. Informe y forma de entrega

Para este trabajo práctico no deberán entregar un informe. En cambio, deberán entregar el mismo contenido que el dado para realizarlo, habiendo modificado los archivos `string_processor.asm`, `string_processor.c` y `main.c`.

La fecha de entrega de este trabajo es 12/04. 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.

Ante cualquier problema con la entrega, comunicarse por mail a la lista de docentes `orga2-doc@dc.uba.ar`.

Por favor no comparta código con compañerxs, ante cualquier incoveniente o demora comuníquese con su tutorx o con los JTPs.