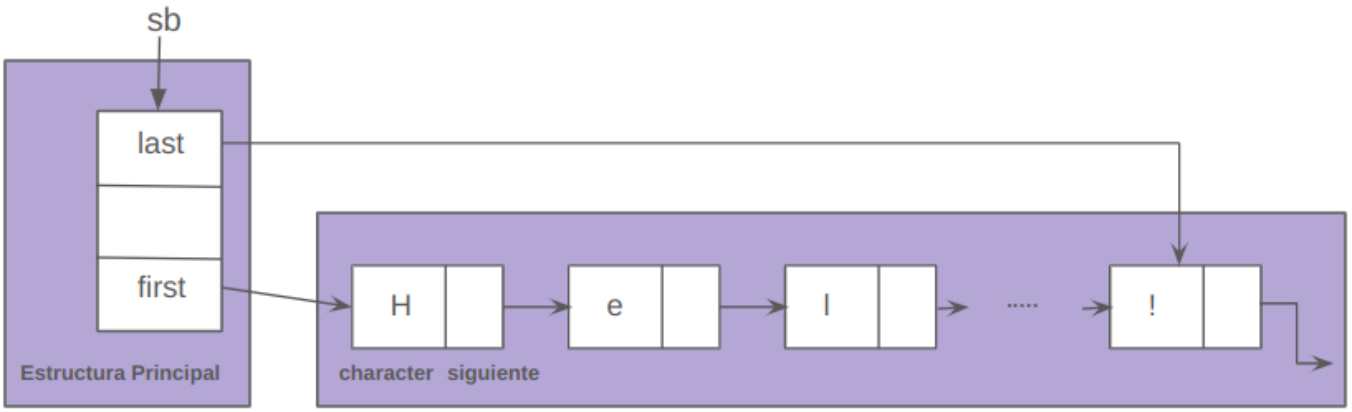
Algoritmos y Estructuras de Datos II

Parcial 31-05: Tema D - TAD: StringBuffer

Se va a implementar un Tipo Abstracto de Datos que representa una secuencia mutable de caracteres. Un *string buffer* es como el TAD string que vimos en los laboratorios, pero puede ser modificado sin crear una nueva instancia. En este examen, el TAD StringBuffer debe implementarse con una estructura principal que tiene entre otros campos un puntero al primer nodo y un puntero al último nodo. El tipo se llama **`stringbuffer`** y si se tiene una variable **`sb`** de ese tipo la representación se puede ver en el siguiente esquema:



Para representar el string vacío, ambos punteros **`first`** y **`last`** de la estructura principal apuntarán a **`NULL`**.

Las operaciones del TAD se listan a continuación:

Función	Descripción
`stringbuffer stringbuffer_empty(void)`	Crea un nuevo string vacío.
`stringbuffer stringbuffer_create(const char *word)`	Crea un nuevo string no vacío que contiene los mismos caracteres que word.
`bool stringbuffer_is_empty(stringbuffer sb)`	Indica si el string **sb** es vacío o no.
`stringbuffer stringbuffer_append(stringbuffer sb, const char c)`	Agrega un carácter **`c`** al final del string **sb** .
`char stringbuffer_char_at(stringbuffer sb, unsigned int index)`	Devuelve el carácter de la posición especificada **index** en el string no vacío **sb**. Index deber ser menor que el largo del string sb.
`stringbuffer stringbuffer_remove(stringbuffer sb, unsigned int index)`	Elimina el carácter de la posición especificada **index** en el string **sb**. Si Index es mayor o igual que el largo del string, no se modifica.

`stringbuffer stringbuffer_replace(`
`  stringbuffer sb,`
`  const char c,`
`  unsigned int index)`
	Reemplaza el carácter de la posición especificada **index** en el string **sb**. Index deber ser menor que el largo del string sb.

`unsigned int stringbuffer_length(` `  stringbuffer sb)`	Devuelve la cantidad de caracteres que contiene el string **sb**.
`char* stringbuffer_to_string(stringbuffer sb)`	Devuelve un arreglo en memoria dinámica que contiene todos los caracteres de **sb**.
`void stringbuffer_dump(stringbuffer sb)`	Muestra todos los caracteres de **sb** en la salida estándar.
`stringbuffer stringbuffer_destroy(` `  stringbuffer sb)`	Destruye el string **sb** liberando toda la memoria utilizada por la instancia.

Se debe lograr que la implementación del TAD garantice que la funciones **stringbuffer_length()** sea de orden constante O(1). Además **el programa resultante no debe dejar *memory leaks ni lecturas/escrituras inválidas*.**

En el archivo **stringbuffer.h** se encuentran descripciones de las funciones enumeradas anteriormente y además se especifican las pre y post condiciones para cada una. Se deben verificar las pre y post usando **assert()** así como también deben definir una *invariante de representación*, aunque sea una básica pero no trivial. **La invariante también debe verificarse en las pre y post condiciones que correspondan.**

Se provee un programa **testing.c** que implementa una suite de tests para poder probar las funciones del TAD. Una vez compilado el programa puede probarse ejecutando:

```
$ ./teststrbuff
```

Consideraciones:

- Solo se debe modificar el archivo **stringbuffer.c**
- Se incluyen signaturas de un par de funciones **static** que creemos pueden ser útiles a la hora de completar el TAD. No es necesario que las implementen, pero hacerlo puede facilitar la tarea.
- Se provee el archivo **Makefile** para facilitar la compilación.
- Se recomienda usar las herramientas **valgrind** y **gdb**.
- Si el programa no compila, no se aprueba el parcial.
- Los *memory leaks* bajan puntos
- Entregar código muy improlijo puede restar puntos
- Si **stringbuffer_length()** no es de orden constante *O(1)* baja muchísimos puntos
- Si en la función **stringbuffer_create()** se necesitara saber el tamaño de la cadena **word**, se puede usar la función **strlen()** de **<string.h>** sin embargo se puede implementar sin conocer la cantidad de caracteres en **word**.
- Es deseable (y resultará más sencillo también) que la operación **stringbuffer_append()** sea de orden constante *O(1)*
- Mientras más elaborada sea la invariante de representación más puntos se suma.
- **No modificar el .h puesto que solo se entregará stringbuffer.c**