# Proyecto 2 - Diccionario y Listas Enlazadas

Algoritmos y Estructuras de Datos II - Laboratorio

**Docentes:** Leonardo Rodríguez, Gonzalo Peralta, Diego Dubois, Jorge Rafael,

## Objetivo

Aprender a implementar tipos abstractos de datos (TADs) en el lenguaje de programación C, asimilando el concepto de ocultamiento de información (link en wikipedia).

Para ello, habrá que:

- Implementar en C los tipos abstractos `map` y `dict` (usando punteros a estructuras y manejo dinámico de memoria).
- Implementar en C una interfaz de línea de comando para que usuarios finales puedan usar el diccionario.
- Reutilizar código objeto dado por la cátedra, para lograr la construcción del ejecutable final.

Se dictarán clases de:

- Punteros y Memoria Dinámica en C.
- TADs en C.
- Listas Enlazadas en C.
- Makefile.

## Mapeo

### Descripción

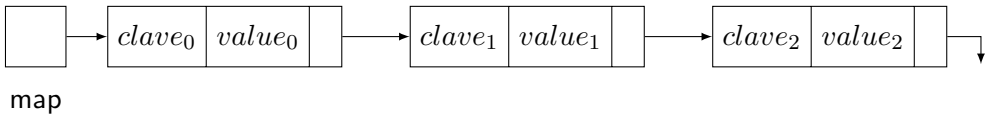
Un mapeo (o *map*) almacena asociaciones de pares clave-valor:

$$\begin{array}{lcl} clave_0 & \mapsto & valor_0 \\ clave_1 & \mapsto & valor_1 \\ \vdots & \vdots & \vdots \\ clave_n & \mapsto & valor_n \end{array}$$

No puede contener claves repetidas. Las operaciones principales que provee este TAD son: agregar una clave junto con su valor (`map_put`), buscar el valor asociado a una clave (`map_get`) y borrar una clave (`map_remove`).

### Implementación

En este proyecto implementaremos el mapeo usando una lista enlazada de nodos con clave y valor.




El mapeo vacío se representará con el puntero `NULL`. Recordar que no puede haber claves repetidas. Tener en cuenta que si se intenta insertar una clave ya existente, `map_put` debe reemplazar el valor asociado a la misma. Revisar `map.h` para ver la descripción completa de las operaciones a implementar. La definición de la estructura del nodo debe ir en `map.c` y no en `map.h`.

**Ninguna** de las funciones debe implementarse de manera recursiva.

## Diccionario

### Descripción

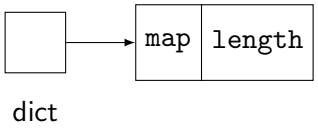
El diccionario (o *dict*) almacenará palabras junto con sus definiciones:

$$\begin{array}{lcl} palabra_0 & \mapsto & def_0 \\ palabra_1 & \mapsto & def_1 \\ \vdots & \vdots & \vdots \\ palabra_n & \mapsto & def_n \end{array}$$

Las operaciones principales del TAD son: agregar una palabra junto con su definición (`dict_add`), buscar la definición de una palabra (`dict_search`), borrar una palabra (`dict_remove`) y conocer la cantidad de palabras disponibles (`dict_length`).

### Implementación

El diccionario se implementará con un puntero a una estructura, que contendrá un mapeo y la longitud actual del diccionario (cantidad de palabras disponibles).




El diccionario vacío se representa con un puntero a una estructura con un mapeo vacío y longitud 0. Revisar `dict.h` para ver la descripción de las operaciones a implementar.

La definición de la estructura debe ir en `dict.c` y no en `dict.h`.

## Interfaz

Además de los TADs deberán implementar una interfaz de línea de comandos (menú de opciones). El menú deberá contener opciones para agregar, buscar y borrar palabras, entre otras.

### Esqueleto de código

La cátedra provee un esqueleto de código con los siguientes archivos:

```
amd64                -- carpeta con código objeto (.o) para arquitecturas de 64 bits.
dict_helpers.c .h    -- funciones para leer y escribir diccionarios desde archivos.
dict-test.c          -- programa que testea las funciones del diccionario.
helpers.c .h         -- funciones para leer strings desde el teclado (útil para el menú).
i386                 -- carpeta con código objeto (.o) para arquitecturas de 32 bits.
map.h                -- descripción del tipo map, funciones a implementar en map.c.
dict.h               -- descripción del tipo dict, funciones a implementar en dict.c.
input                -- carpeta con archivos de texto con palabras y definiciones.
string.c .h          -- código del tipo string.
```

Deben implementar los archivos `dict.c`, `map.c` y `main.c`. Los archivos del esqueleto **no** se pueden modificar.

#### Código objeto:

En la carpeta `amd64` pueden encontrar el código objeto de cada TAD y de la interfaz. Esos archivos `.o` fueron generados mediante la compilación de los archivos del proyecto ya resuelto por los profesores. Podrán usar el código objeto, pero no podrán ver el código C que los generó. El propósito de estos archivos es que puedan desarrollar el proyecto por partes, por ejemplo: desarrollar tu propio `dict.c` pero usar `map.o` y `main.o` de la cátedra, o desarrollar `main.c` pero usar `map.o` y `dict.o` de la cátedra. Eventualmente los tres archivos deben ser programados por ustedes y no se deben usar más los códigos objetos provistos por los profesores.

## Tareas

Se recomienda realizar las siguientes tareas *en orden*:

1. Bajar el esqueleto.
2. Probar el diccionario hecho por los profesores. Para ello situarse en la carpeta donde bajaron el esqueleto y escribir:

   ```
   $ gcc -Wall -Werror -Wextra -pedantic -std=c99 -c helpers.c dict_helpers.c string.c
   $ gcc -Wall -Werror -Wextra -pedantic -std=c99 -o dictionary \
   helpers.o dict_helpers.o string.o amd64/*.o
   ```

   La primera línea compila cada archivo `.c` listado y genera un archivo `.o` con el mismo nombre. La segunda línea enlaza todos los archivos `.o` generados en la línea anterior más los que están en la carpeta `amd64` para generar un ejecutable de nombre `dictionary`.

   ```
   $./dictionary
   ```

3. Implementar el TAD `dict` usando `amd64/main.o` y `amd64/map.o`. Crear un archivo de nombre `dict.c`, definir allí la estructura y programar las funciones descriptas en `dict.h`. Para ello deben usar las funciones de `map.h` y `dict_helpers.h`. Una vez que terminaron, pueden probarlo de la siguiente forma:

   ```
   $ gcc -Wall -Werror -Wextra -pedantic -std=c99 -c dict.c
   $ gcc -Wall -Werror -Wextra -pedantic -std=c99 -o dictionary \
   helpers.o dict_helpers.o string.o dict.o amd64/main.o amd64/map.o
   ```

Notar que **no** usamos `amd64/dict.o` (generado por los profesores), sino que usamos `dict.o` generado por la primera línea. Si usáramos ambos dará error de enlazado por duplicación de definiciones. Ejecutar el nuevo programa:

```
$./dictionary
```

4. Implementar el TAD `map` usando `amd64/main.o`. Crear un archivo de nombre `map.c`, definir allí la estructura del nodo e implementar las funciones descriptas en `map.h`. Una vez terminado, compilar y ejecutar de la siguiente forma:

   ```
   $ gcc -Wall -Werror -Wextra -pedantic -std=c99 -c map.c
   $ gcc -Wall -Werror -Wextra -pedantic -std=c99 -o dictionary \
   helpers.o dict_helpers.o string.o dict.o map.o amd64/main.o
   $ ./dictionary
   ```

   Notar que **no** usamos `amd64/dict.o` ni `amd64/map.o`.

5. Implementar la interfaz. Crear un archivo llamado `main.c` y allí programar un menú de opciones como el que brinda `amd64/main.o`. **Es importante** respetar las letras de cada opción ('a' para agregar, 'r' para borrar, etc) para facilitar la corrección del proyecto con herramientas automáticas. Usar las funciones de `helpers.h` para leer las opciones y las palabras del teclado. Una vez terminado, compilar y ejecutar:

   ```
   $ gcc -Wall -Werror -Wextra -pedantic -std=c99 -c main.c
   $ gcc -Wall -Werror -Wextra -pedantic -std=c99 -o dictionary \
   helpers.o dict_helpers.o string.o dict.o map.o main.o
   $ ./dictionary
   ```

   Notar que ya no usamos ningún `.o` de la carpeta `amd64`.

6. Tratar de romper el diccionario *a toda costa*, encontrar errores y arreglarlos. ¿Qué pasa si busco una palabra que no está? ¿Y si busco en un diccionario vacío? ¿Y si agrego una palabra repetida?.

En caso de usar una computadora de 32 bits, usar la carpeta `i386` en vez de `amd64`. En caso de usar MAC, solicitar a los profesores una carpeta apropiada. El laboratorio tiene computadoras `amd64`. Se puede usar el comando `uname -r` para conocer la arquitectura de la computadora.

### Chequear pérdidas de memoria

Los programas deben estar libres de *memory leaks* (utilizar el programa `valgrind` para comprobar esto). El comando para usar `valgrind` es:

```
$ valgrind --leak-check=full --show-reachable=yes ./dictionary
```

(notar que para obtener información más exacta, `valgrind` requiere que se compilen los código fuentes con la opción `-g`).

### Testing de unidad

Pueden compilar y ejecutar el archivo `dict-test.c` para testear el TAD dict. Esto **no** testea la interfaz, sólo prueba `dict.c`. Pasar todos los tests no significa estar aprobado. No pasar un test no significa estar reprobado.

### Makefile

En clase veremos cómo crear un archivo `Makefile` que nos ayudará a compilar el proyecto de forma más fácil y organizada.

## Punto estrella

Modificar la implementación de `map_put` para que las claves se mantengan ordenadas (alfabéticamente) de forma invariante. Cada vez que se ingresa una clave nueva, la función debe decidir entre qué nodos insertarla para mantener el orden alfabético (mientras más a la derecha está el nodo, mayor es la clave). Pensar cómo optimizar la funciones `map_get` y `map_remove` para aprovechar esta nueva invariante (¿qué pasa si busco una clave que no está?).

## Entrega

- Fecha de entrega: hasta 26 de Abril, 14hs. Seguir las instrucciones enviadas por mail para la entrega.
- Archivos a entregar: `main.c`, `dict.c` y `map.c`.
- Parcialito: Se tomará programar una función nueva para el TAD `map` y un `main.c` sencillo para probarla.

## Recordar

- Preguntar en clase o en la lista de mails las dudas. En caso de mandar un mail, no poner código.
- Comentar e indentar el código apropiadamente.
- Todo el código tiene que usar la librería estándar de C, y no se puede usar extensiones GNU de la misma.
- El programa resultante **no** debe tener *memory leaks* **ni** accesos (read o write) inválidos a la memoria.
- Recordar que la traducción de pseudocódigo al lenguaje C **no** es directa. En particular, tener en cuenta que lo que se llama `tuple` en el teórico, en C es un `struct`.