# Proyecto 2 - Diccionario con Listas Enlazadas

Algoritmos y Estructuras de Datos II - Laboratorio

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

## Objetivo

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

Para ello, habrá que:

- Implementar en C el TAD `dictionary` (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.

## Instrucciones

En este proyecto se deberá implementar en C un diccionario análogo al del proyecto de la materia Algoritmos y Estructuras de Datos I (en donde se implementó un diccionario sobre lista de asociaciones en el lenguaje Haskell).

### Qué es un `diccionario`?

En computación, un diccionario (o *mapping*) es un contenedor de datos que mapea claves a valores. Ejemplos de la vida real de diccionarios:

- diccionarios de palabras (mapean una palabras a su definición)
- las guías de teléfonos (mapean un nombre a un teléfono)

De lo arriba expuesto se deduce que hay varios tipos de datos involucrados en un diccionario:

- El diccionario en sí, que contiene todos los mapeos
- Las claves
- Los valores

En este proyecto, se implementará un diccionario usando una lista de asociación, y cada elemento de esta lista serán pares de clave y valor.

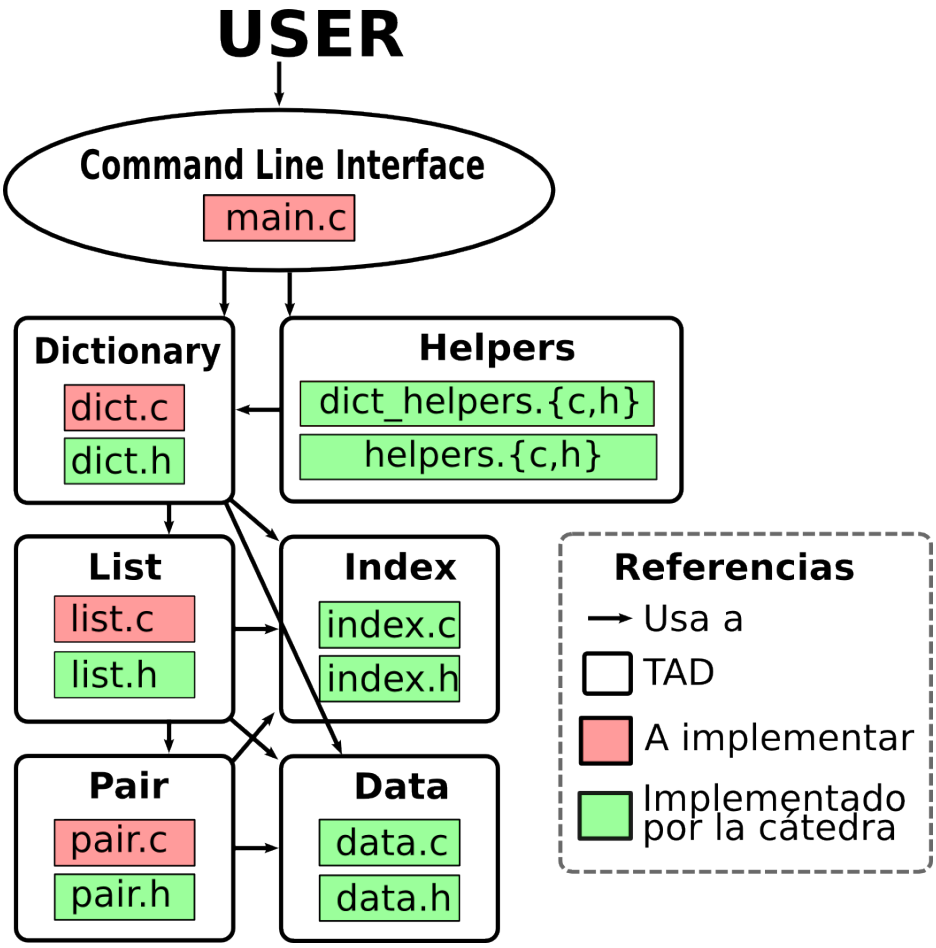


Figura 1: Diagrama de TADs

### Estructura del proyecto

En la figura 1, se muestra un diagrama de los tipos abstractos de datos necesarios para la resolución de este proyecto. La lista de asociación se llamará `list`, y los pares de clave y valor requerirán de 3 TADs: el `pair`, `index` y `data`, respectivamente.

Los módulos `.c` resaltados en **verde** serán provistos por la cátedra, y los módulos resaltados en **rojo**, deberán ser implementados por Uds.

### Códigos objeto

La cátedra también provee para cada archivo en **rojo** el código objeto (.o) correspondiente para que ustedes puedan testear su código por separado. En la siguiente sección explicamos cómo usar esos archivos objeto.

Tener en cuenta que se distribuirán códigos objeto para arquitecturas de 32 y 64 bits, con lo cual Uds. deberán elegir cuál usar en función de la arquitectura y sistema operativo de la computadora que usen.

Todas las computadoras del lab de la facultad tienen arquitectura de **64 bits**.

Para ver qué arquitectura corresponde en una computadora corriendo Linux, abrir una terminal y correr el siguiente comando:

```
$ uname -a
```

El resultado va a mostrar algo como este ejemplo, que muestra que la arquitectura es de 64 bits:

```
Linux foo 3.2.0-39-generic ... UTC 2013 x86_64 x86_64 x86_64 GNU/Linux
```

Si alguien va a usar una computadora con Mac OS, tiene que avisar usando la lista de mail para que le generemos código objeto compatible.

## Primera tarea: probar el diccionario

Como el esqueleto tiene todos los archivos `.o`, podemos generar el ejecutable para ver cómo debe quedar el proyecto una vez terminado.

Seguir los siguientes pasos:

- Compilar los .o de todos los .c dados:

  ```
  $ gcc -Wall -Werror -Wextra -pedantic -std=c99 -c dict_helpers.c
  $ gcc -Wall -Werror -Wextra -pedantic -std=c99 -c helpers.c
  $ gcc -Wall -Werror -Wextra -pedantic -std=c99 -c index.c
  $ gcc -Wall -Werror -Wextra -pedantic -std=c99 -c data.c
  ```

  Esto generó cuatro nuevos archivos:

  ```
  dict_helpers.o helpers.o index.o data.o
  ```

- Enlazar esos archivos `.o` junto con los dados por la cátedra para generar el ejecutable final.

  ```
  $ gcc  -Wall -Werror -Wextra -pedantic -std=c99 -o dictionary *.o amd64/*.o
  ```

  Si estamos en una arquitectura de 32bits, reemplazar `amd64` por `i386` en el comando anterior.

- Ejecutar el diccionario:

  ```
  $ ./dictionary
  ```

  Elegir la opción `1` para cargar el diccionario desde un archivo, en la carpeta `input` hay dos diccionarios para probar.

## Implementación

La manera recomendada para implementar el proyecto es *"de arriba para abajo"* (mirando el esquema de la figura 1). Son en total 4 archivos a implementar:

```
main.c dict.c list.c pair.c
```

Para implementar cada TAD se deberá tener en cuenta:

- Un TAD opaco en C es una librería, y consta de dos archivos separados, un `.h` ("header" o cabecera) y un `.c` (la lógica e implementación per se).
- Todos los TAD's deberán ser implementados con la técnica de punteros a estructuras que se va a presentar en el teórico del laboratorio.
- Para implementar correctamente un TAD, el `.h` debe exportar **únicamente** las funcionalidades que la interfaz del TAD define, ocultando todos los aspectos que tienen que ver con su implementación (como por ejemplo, la definición de la estructura en sí, que es un detalle de implementación y **debe** permanecer oculto).

- Compilar todos los archivos (y enlazar el ejecutable final) con las flags usuales:

  `-Wall -Werror -Wextra -pedantic -std=c99 -g`

  (notar que la opción `-g` es necesaria para luego poder utilizar programas como `gdb` y `valgrind`).

  Es decir, que para compilar los códigos fuentes hay que correr el comando:

  ```
  $ gcc -Wall -Werror -Wextra -pedantic -std=c99 -g -c *.c
  ```

  y luego, para enlazar el ejecutable final:

  ```
  $ gcc -Wall -Werror -Wextra -pedantic -std=c99 -g -o dictionary *.o.
  ```

#### Probar por separado

También se puede enlazar con los `.o` de la cátedra con el fin de testear la implementación. Por ejemplo, supongamos que de los 4 archivos `.c` sólo tenemos listo nuestro `main.c`, y ahora queremos testearlo usando los otros 3 archivos `.o` de la cátedra. Entonces hacemos

```
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 -g -c *.c
```

para compilar y luego

```
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 -g -o dictionary *.o \
amd64/dict.o amd64/list.o amd64/pair.o
```

para enlazar.

Notar que en el comando anterior **no** utilizamos el archivo `amd64/main.o` pues estamos usando nuestro propio `main.o` que fue generado en la compilación de nuestro `main.c`. No se pueden usar los dos `main.o` al mismo tiempo.

Se puede ejecutar un comando similar para probar los otros archivos, siempre teniendo en claro cuándo estamos usando nuestro código objeto y cuándo el de la cátedra.

La idea es que cuando todos los archivos `.c` estén listos ya no necesitemos usar **ningún** archivo `.o` de la cátedra.

#### 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`).

#### Makefile

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

### El esqueleto de código

Resumiendo, el esqueleto tiene los siguientes archivos:

**main.o**

El módulo que implementa la interfaz de línea de comandos. Puede ser usado para ejecutar el diccionario sin tener implementado el `main.c` propio, pero solo de manera **temporal**. Cada grupo debe implementar su `main.c` y generar el código objeto desde ahí.

**dict.h, dict.o**

La librería que representa un **diccionario**. Se entrega además el `dict.o` para que usen de manera temporal, cada grupo debe implementar su propio `dict.c` y así generar el propio código objeto.

**list.h, list.o**

La librería que representa una lista enlazada. Es el TAD más complejo del proyecto.

Este TAD lista será el contenedor de pares de claves y valores del diccionario.

**pair.h, pair.o**

Librería que representa un par de elementos, es decir, una 2-upla. El primer valor es de tipo `index` y el segundo de tipo `data`).

**index.h, index.c**

La librería que abstrae el tipo de dato **clave**, que se usa para guardar las palabras a definir en el diccionario. Se entrega el código fuente para que sea estudiado y analizado por los alumnos.

**data.c, data.h**

La librería que abstrae el tipo de dato **valor**, que se usa para guardar las definiciones del diccionario. Se entrega el código fuente para que sea estudiado y analizado por los alumnos.

**Helpers**

Se entregan además dos librerías que proveen funciones que ayudan a manipular diccionarios desde y hacia archivos, y leer input de usuario de largo arbitrario desde la entrada estándar:

```
dict_helpers.c
dict_helpers.h

helpers.c
helpers.h
```

También en este caso se entrega el código fuente para que sea estudiado y analizado por los alumnos.

Continuamos ahora con los detalles de la implementación.

## Detalles de implementación

### TAD `dict`

Observar el archivo `dict.h` donde se detalla la interfaz del TAD diccionario. Se incluyen comentarios, precondiciones y postcondiciones, que deberán ser respetados en la implementación (en el `dict.c`).

Tener en cuenta que hay que utilizar las bibliotecas `index.h`, `data.h`, `list.h` y `pair.h`.

En este proyecto se espera que se implemente el TAD dict usando listas enlazadas para su representación interna (y oculta). Por lo cual, pensar qué miembros (y de qué tipo sería cada miembro) necesitaría tener la estructura (oculta) del dict.

Es decir, en el `dict.c`, deberían tener algo de la pinta:

```
struct _dict_t {
    list_t data;
     /* algo más? Pensar! Tener en cuenta los órdenes de las operaciones */
};
```

### Interfaz de usuario

Desarrollar una interfaz de línea de comando similar a la que se utilizó para el proyecto de Algoritmos I. Para esta implementación sólo se deben utilizar las funciones y los tipos exportados `dict.c`, más las funciones auxiliares provistas en la biblioteca `helpers.h`.

La interfaz deberá ofrecer las siguientes operaciones, respetando las letras para cada opción:

- **z** Mostrar el tamaño del diccionario en uso.
- **s** Buscar una definición para una palabra dada (en el diccionario en uso).
- **a** Agregar una palabra con su correspondiente definición al diccionario (ídem).
- **d** Borrar una palabra (y su definición, ídem).
- **e** Vaciar el diccionario actual.
- **h** Mostrar el diccionario actual por la salida estándar.
- **c** Duplicar el diccionario actual, mostrando el diccionario copiado por la salida estándar.
- **l** Cargar un nuevo diccionario desde un archivo dado por el usuario.
- **u** Guardar el diccionario actual en un archivo elegido por el usuario.
- **q** Finalizar.

El menú arriba descripto debe ser mostrado al usuario de manera cíclica hasta que se elija la opción de finalizar.

Tener en cuenta que hay opciones (como las **s**, **a**, **d**), que requieren una interacción extra con el usuario. Por ejemplo, para buscar una palabra en el diccionario, se deberá pedir al usuario que se ingrese la palabra a buscar, luego leer lo que el usuario ingrese, y usar ese input para obtener un resultado (que luego debe ser mostrado por standard output). Para facilitar la tarea de leer el input del usuario, se provee en la biblioteca `helpers` (provista por la cátedra) una función auxiliar `readline_from_stdin`.

Por ejemplo, para leer una línea e imprimirla por pantalla tendríamos algo como:

```
#include <stdio.h>
#include <stdlib.h>

#include "helpers.h"

int main(void) {
    char *one_line = NULL;

    printf("Introduzca un mensaje: ");
    one_line = readline_from_stdin();
    printf("Su mensaje: %s\n", one_line);

    free(one_line);
    one_line = NULL;
    return (0);
}
```

Para implementar la opción **h** (mostrar el diccionario actual por la salida estándar), deberán, primero que nada, leer la página de manual para `stdout`:

```
$ man stdout
```

En esa página de manual podrán leer cómo la constante `stdout` (provista en la biblioteca `stdio.h`) es el valor de tipo `FILE *` necesario para usar como segundo argumento al llamar a `dict_dump`, logrando así que el diccionario sea impreso en la pantalla.

Asimismo, para la implementación de las penúltimas dos opciones (**l** y **u**), se proveen otras dos funciones auxiliares: `dict_from_file` y `dict_to_file`. Se recomienda utilizarlas ya que facilitan la tarea.

Para referencia, ver el archivo `dict_helpers.h` (entregado en el esqueleto de código provisto por la cátedra) que documenta cada una de las funciones nombradas arriba.

### TAD `pair`

El TAD `pair` a implementar en C es equivalente a la siguiente definición en pseudo-código de una 2-upla que almacena como primer valor algo de tipo `index_t`, y como segundo valor algo de tipo `data_t`:

```
(index_t, data_t)
```

Es decir, que la estructura `struct _pair_t` deberá ser elegida de manera tal que se pueda almacenar dos miembros: un `index_t` y un `data_t`.

Para manipular los index y los data hay que usar todos los métodos documentados en los archivos `index.h` y `data.h`.

### TAD `list`

El TAD `list` a implementar en C es equivalente a la siguiente definición en pseudo-código (pensar en Haskell, lista de pares de clave-valor):

```
[(index_t, data_t)]
```

que dada la definición del tipo nuevo `pair_t` de la subsección anterior, se puede escribir de manera equivalente:

`[pair_t]`

Es decir, que la estructura `struct _list_t` deberá ser elegida de manera tal que se puedan almacenar nodos que contengan un `pair_t` adentro.

Para definir la estructura del TAD `list`, seguir el apunte del teórico sobre listas enlazadas y mapear idénticamente a C los dos tipos definidos como `node` y `list`. Notar que el tipo `node` es un detalle de implementación de la lista, y no debe ser expuesto bajo ningún concepto en el archivo de cabecera `list.h`.

Si uno piensa cada nodo como una figura geométrica, y cada puntero como una flecha, la lista de pares que tienen que implementar puede graficarse como en la figura 2.

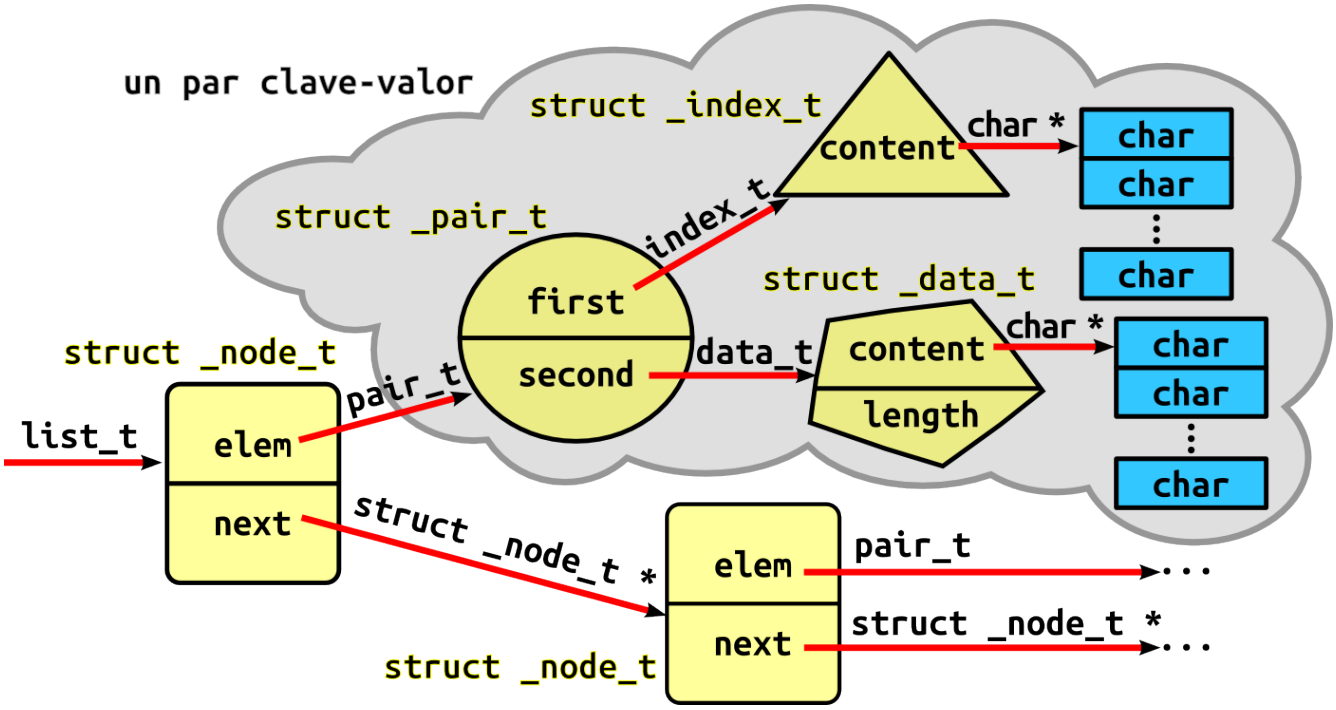


Figura 2: La lista enlazada de pares clave-valor

Todas las figuras en amarillo son estructuras de C, y todas las flechas rojas son punteros (direcciones de memoria).

Las cajas rectangulares con miembros **elem** y **next** son nodos de la lista, y la nube completa representa un par clave-valor completo. Entonces, el miembro `next` de un nodo apunta a un par clave-valor, y el miembro `next` apunta al nodo siguiente de la lista. Si un nodo es el último de la lista, `next` es un puntero nulo (es decir, vale `NULL`).

Notar que la representación de la lista en sí es **únicamente** un puntero al primer nodo, **y nada más**.

Dentro del par clave-valor, el círculo es la estructura del TAD `pair`, el pentágono es un valor (representa al TAD `data`) y el triángulo una clave (representa al TAD `index`).

Notar además que cada index y data tienen, a su vez, memoria asociada para almacenar sus contenidos (son las cajitas rectangulares celestes).

## Ejercicios estrella

Como siempre, los puntos estrella son opcionales. Se recomienda fuertemente no hacerlos hasta tener todo lo demás listo.

*En este proyecto, cada punto estrella debe entregarse en una carpeta separada*.

**Ejercicio ⋆ 1** *Lista ordenada*

La tarea es modificar la implementación de la lista de tal forma que la misma permaneza ordenada en todo momento.

Recordemos que la función `list_append` inserta siempre al final de la lista, sin tener el cuenta el orden de la misma. Por lo tanto necesitamos reemplazarla por una nueva función `list_add` que inserte los pares en la posición adecuada.

Agregar un nueva función:

```
list_t list_add(list_t list, index_t index, data_t data);
```

y eliminar la función `list_append`.

También hay que modificar `list_search` y `list_remove`, ¿Cómo se puede aprovechar en estas funciones el hecho de que la lista está ordenada?.

Notar que **no** hay que implementar ningún algoritmo de ordenación.

**Ejercicio ⋆ 2** *Lista enlazada con puntero al último nodo*

Modificar la implementación para que la lista mantenga un puntero al último nodo (además del primero).

Luego modificar la función `list_append` para que tenga orden constante en vez de lineal, y corregir también el resto de las funciones apropiadamente.

Ayuda: Una lista deberá ser implementada con un puntero a una estructura con dos miembros, el primero y el último nodo. Tener cuidado: la representación de una lista vacía ya no es el puntero nulo.

(Este punto es incompatible con el punto estrella anterior, hacerlos por separado).

## Entrega y evaluación

- Fecha de entrega del código y parcialito: Martes 21 de Abril.

## Recordar

- 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`.