Proyecto 2 - Tipos Abstractos de Datos

Algoritmos y Estructuras de Datos II - Laboratorio

Docentes: Natalia Bidart, Matías Bordese, Diego Dubois, Leonardo Rodríguez.

1. 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.
- Seguir utilizando regularmente un sistema simplificado de "Integración Continua" para obtener un diagnóstico temprano del proyecto (Jaime).

2. 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).

2.1. 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.

2.2. Implementación

En la figura 1, se muestra un diagrama de los tipos abstractos de datos necesarios (listados en la sub-sección anterior) 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.

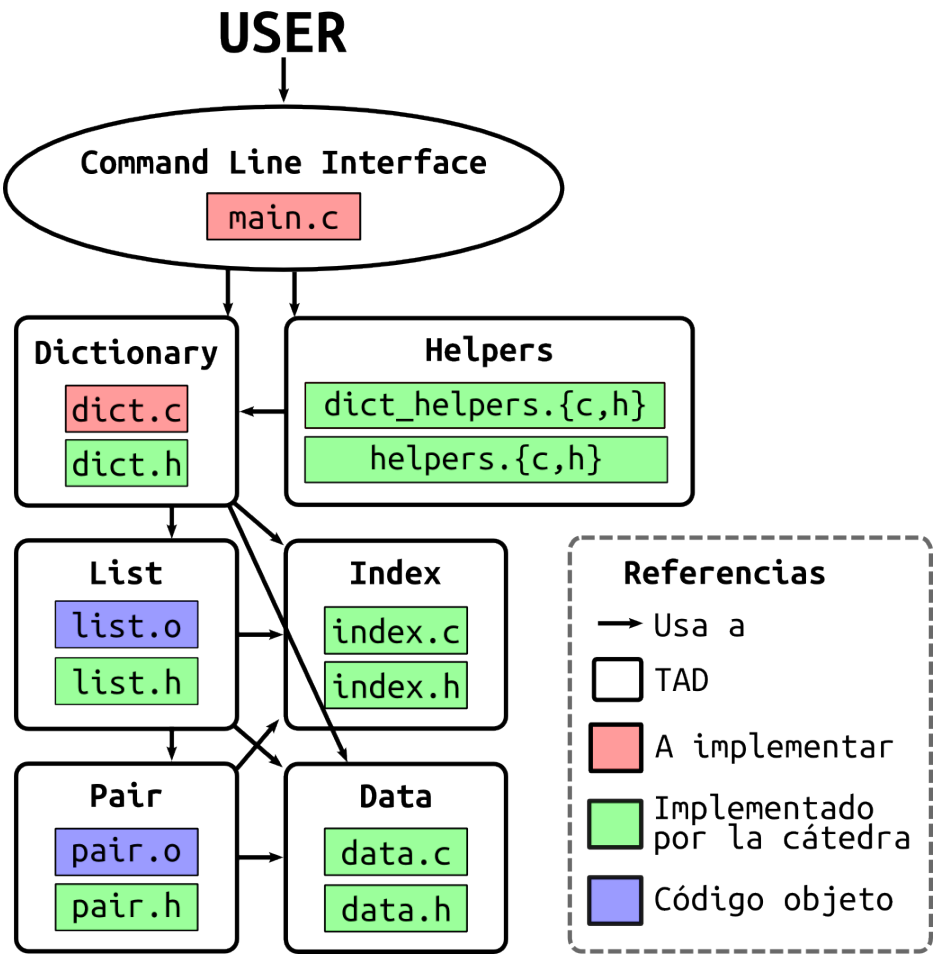

Figura 1: Diagrama de TADs

Los módulos `.c` resaltados en **verde**, y los códigos objeto (`.o`, en **azul**) serán provistos por la cátedra, y los módulos resaltados en **rojo**, deberán ser implementados por Uds.

Para implementar los archivos `dict.c` y `main.c`, se deberán usar los archivos de cabecera `list.h` y `pair.h` (con sus correspondientes códigos objecto), para poder así compilar y construir el ejecutable final. 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 requieren usar los `list.o` y `pair.o` para **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.

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 linkear 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 de debugging y chequeo de memoria del código, como se muestra en el próximo ítem).

  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 linkear el ejecutable final:

  ```
  $ gcc -Wall -Werror -Wextra -pedantic -std=c99 -g -o dictionary *.o
  ```
- 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`).
- Se recomienda no empezar a resolver los puntos estrella hasta que no se terminen las implementaciones obligatorias en su totalidad, y que se pruebe el proyecto confirmando que se cumplen todos los requerimientos arriba listados.

2.3. El esqueleto de código

La cátedra provee un esqueleto de código que puede ser usado sin ninguna modificación y sin ningún agregado extra para probar el diccionario. Para ello, bajar el esqueleto, descomprimirlo en una carpeta y, dentro del directorio recién creado, compilar usando los comandos:

- 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
  ```
- Generar el ejecutable final usando los .o recién creados más los dados:

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

El esqueleto tiene los siguientes archivos:

2.3.1. 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.

2.3.2. dict.h, dict.o

La librería que representa un **diccionario** (estilo Larousse, con tapas y hojas de papel). 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.

2.3.3. list.h, list.o

La librería que representa una lista enlazada. Es el TAD más complejo del proyecto, y por ahora no debe ser implementado por los alumnos, sino que deben re-usar el `list.o` dado en el esqueleto.

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

2.3.4. 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` (ver abajo).

Es un TAD muy muy simple, y por ahora no debe ser implementado por los alumnos, sino que deben re-usar el `pair.o` dado en el esqueleto.

2.3.5. 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.

2.3.6. 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.

2.3.7. 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.

2.4. Detalles de las tareas

1. A continuación se detalla la interfaz del **único** TAD a implementar en esta primer parte del proyecto. 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`.

```
#ifndef _DICT_H
#define _DICT_H

#include <stdio.h>
#include <stdbool.h>


typedef char *word_t;
typedef char *def_t;
typedef struct _dict_t *dict_t;

dict_t dict_empty(void);
/*
 * Return a newly created, empty dictionary.
 *
 * The caller must call dict_destroy when done using the resulting dict,
 * so the resources allocated by this call are freed.
 *
 * POST: the result is not NULL, and dict_length(result) is 0.
 */

dict_t dict_destroy(dict_t dict);
/*
 * Free the resources allocated for the given 'dict', and set it to NULL.
 */

unsigned int dict_length(dict_t dict);
/*
 * Return the amount of elements in the given 'dict'.
 * Constant order complexity.
 *
 * PRE: 'dict' is not NULL.
 */

bool dict_is_equal(dict_t dict, dict_t other);
/*
 * Return whether 'dict' is equal to 'other'.
 *
 * PRE: 'dict' and 'other' are not NULL.
 */

bool dict_exists(dict_t dict, word_t word);
/*
 * Return if the given 'word' exists in the dictionary 'dict'.
 *
 * PRE: 'dict' and 'word' are not NULL.
 */

def_t dict_search(dict_t dict, word_t word);
/*
 * Return the definition associated with 'word'.
 *
 * The caller must free the resources allocated for the result when done
 * using it.
```

```
 *
 * PRE: 'dict' and 'word' are not NULL, and 'word' does exist in 'dict'.
 *
 * POST: the result is not NULL.
 */

dict_t dict_add(dict_t dict, word_t word, def_t def);
/*
 * Return the given 'dict' with the given ('word', 'def') added.
 *
 * The given 'word' and 'def' are inserted in the dict,
 * so they can not be destroyed by the caller.
 *
 * PRE: all 'dict', 'word' and 'def' are not NULL, and 'word' does not
 * exist in the given 'dict'.
 *
 * POST: the elements of the result are the same as the one in 'dict' with
 * the new pair ('word', 'def') added.
 */

dict_t dict_remove(dict_t dict, word_t word);
/*
 * Return the given 'dict' with the given 'word' removed.
 *
 * PRE: 'dict' and 'word' are not NULL, and the definition for 'word' does
 * exists in the dictionary.
 *
 * POST: the elements of the result are the same as the one in 'dict' with
 * the entry for 'word' removed.
 */

dict_t dict_copy(dict_t dict);
/*
 * Return a newly created copy of the given 'dict'.
 *
 * The caller must call dict_destroy when done using the resulting dict,
 * so the resources allocated by this call are freed.
 *
 * POST: the result is not NULL and it is an exact copy of 'dict'.
 * In particular, dict_is_equal(result, dict) holds.
 */

void dict_dump(dict_t dict, FILE *fd);
/*
 * Dump the given 'dict' in the given file descriptor 'fd'.
 *
 * PRE: 'dict' is not NULL, and 'fd' is a valid file descriptor.
 */

#endif
```

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 */
```

```
};
```

2. 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.

Punto ⋆ 1 *Búsqueda de palabras más eficiente*

La búsqueda en el diccionario tiene como precondición que la palabra exista. Esto puede producir una pérdida de eficiencia ya que cada vez que se quiera buscar una definición hay que ver antes si la palabra existe. Pensar una forma de solucionar este problema.

Ayuda Modificar la implementación del diccionario para que la estructura interna (el `struct`) soporte guardar la última palabra buscada (algo similar a una *cache* muy simplificada). Notar que este punto estrella debe requerir cambios **únicamente** en el `dict.c`.

3. Entrega y evaluación

- Fecha de entrega del código:
  - Desde el Lunes 7 de Abril a las 00:00 hrs hasta las 23:59 hrs del mismo día.
  - Obligatorio hacerlo a través de Jaime, habrá una tarea específica para subir el código (separada de la tarea para probar los proyectos).
  - Se permitirá una sola entrega por grupo, así que asegurarse de subirla cuando esté lo suficientemente probada.
- Sin parcialito esta primer parte.

4. Recordar

- Comentar e indentar el código apropiadamente, siguiendo el estilo de código ya provisto por la cátedra.
- 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`.