Proyecto 2
Tipos Abstractos de Datos

Algoritmos y Estructuras de Datos II - Laboratorio

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

1. Objetivo

El objetivo de este proyecto es aprender a implementar tipos abstractos de datos (TADs) opacos en el lenguaje de programación C, internalizando el concepto de ocultamiento de información[1].

Para ello, habrá que:

- Implementar en C el TAD `dictionary` (utilizando 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, para lograr la construcción del ejecutable final.

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).

En la figura 1, se muestra un diagrama de los tipos abstractos de datos necesarios para la resolución de este proyecto.

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.

Como referencia, todas las computadoras del lab de la facultad requieren usar los `list.o` y `pair.o` para **64 bits**.

[1] Ver en wikipedia

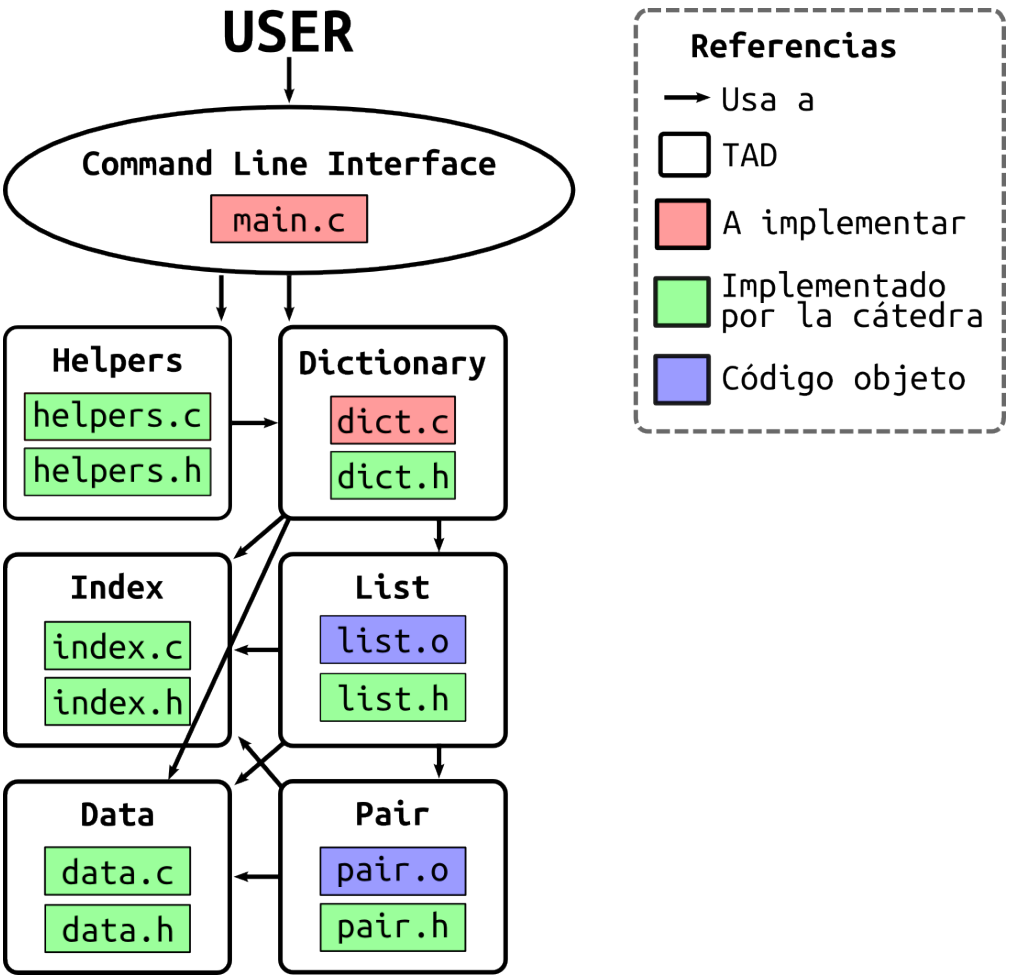

Figura 1: Diagrama de TADs

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
```

Para implementar cada TAD se deberá tener en cuenta:

- Un TAD opaco en C 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 presenta 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 programa con las opciones usuales:

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

  (notar que la opción `-g` es necesaria para luego poder utilizar programas de debugging y chequeo 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). Un ejemplo de 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.

2.1. Detalles de las tareas

1. A continuación se detalla la interfaz del única 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' must not be NULL.
 */

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

bool dict_exists(dict_t dict, word_t word);
/*
 * Return if the given 'word' exists in the dictionary 'dict'.
 *
 * PRE: both 'dict' and 'word' must not be 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: both 'dict' and 'word' must not be NULL, and 'word' must exist in
 * 'dict'.
 *
 * POST: the result is not NULL.
 */

dict_t dict_add(dict_t dict, word_t word, def_t def);
/*
 * Return a dictionary equals to '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' must not be 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 a dictionary equals to 'dict' with the given 'word' removed.
 *
 * PRE: both 'dict' and 'word' must not be NULL, and the definition for 'word'
 * 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' must not be NULL, and 'fd' must be 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 usuarui 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 el módulo `helpers` una función auxiliar `readline_from_stdin`.

Para implementar la opción **h** (mostrar el diccionario actual por la salida estándar), deberán 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 `helpers.h` (incluido a continuación y entregado en el código provisto) que documenta cada una de las funciones nombradas arriba.

```
#ifndef _HELPERS_H
#define _HELPERS_H

#include "dict.h"


char *readline_from_stdin(void);
/*
 * Read the user input from standard input until a newline is detected,
 * and return the corresponding (dinamically allocated) string.
 *
 * The caller to this function is responsible for the allocated memory.
 *
 * POST: A new null-terminated string is returned with the content read from
 *       standard input, or NULL if there was an error.
 */

dict_t dict_from_file(char *filename);
/*
```

```
 * Return a dict instance populated by the data in the given filename.
 *
 * PRE: filename is the path to an existent and accessible file.
 *
 * file data is formatted as follows:
 *      word_1: definition_1
 *      ...
 *      word_N: definition_N
 *
 * POST: Returned dict is not NULL and contains the words and respective
 *       definitions listed in the given file.
 */

void dict_to_file(dict_t dict, char *filename);
/*
 * Write dict data to file with the format expected by dict_from_file.
 *
 * PRE: 'dict' must not be NULL.
 *
 * POST: 'dict' words and respective definitions are written to 'filename'
 *       using the format:
 *       word_1: definition_1
 *       ...
 *       word_N: definition_N
 */

#endif
```

Punto $\star$ 1 *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 soporte guardar la última palabra buscada (algo similar a una* cache *muy simplificada). Notar que este punto estrella debe incluir cambios* **únicamente** *en el* `dict.c`.

2.2. Recordar

- Entrega y evaluación de esta primer parte: Martes 23 de Abril de 2013.
- Comentar e indentar el código apropiadamente, siguiendo el estilo de código ya provisto por la cátedra en los archivos dados.
- 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.