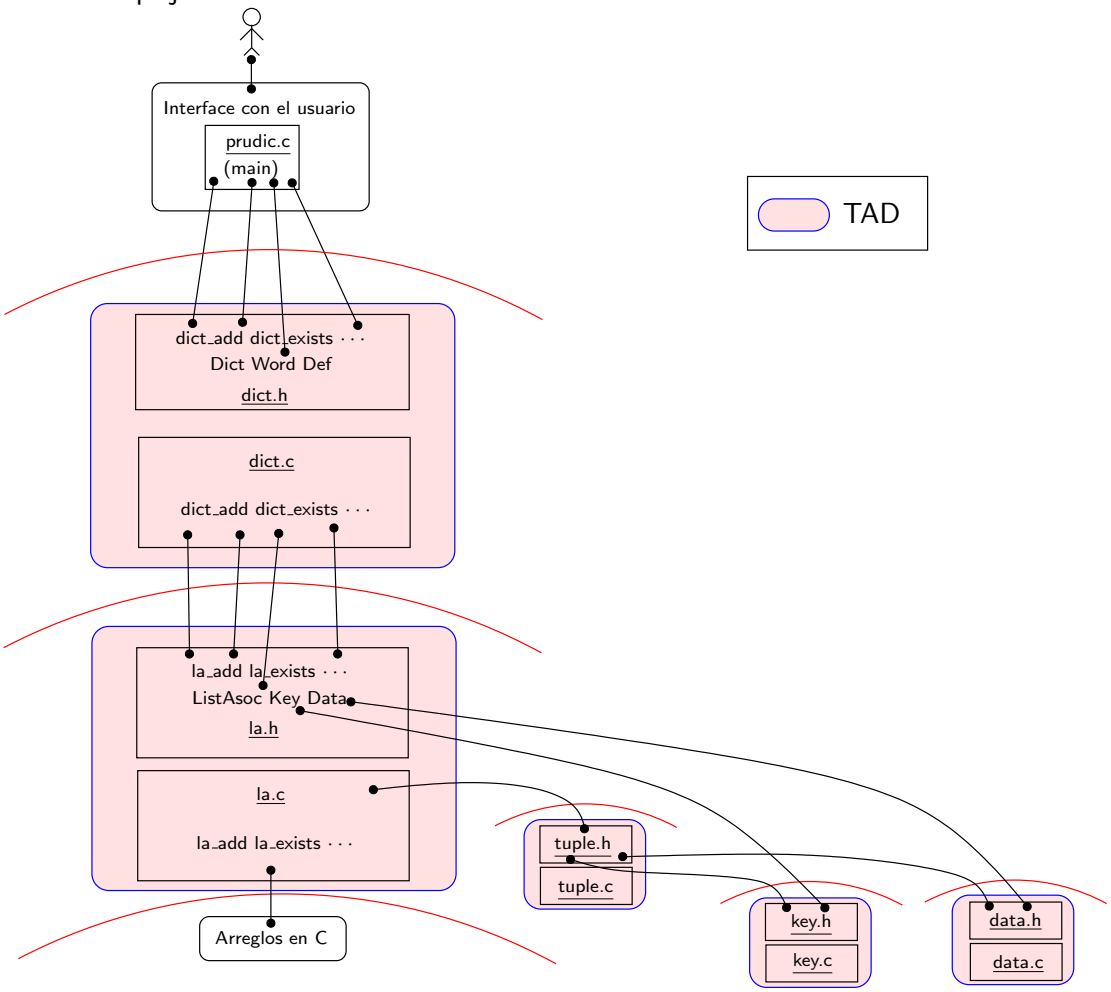
Proyecto 1 parte 2
Diccionario sobre arreglos

Algoritmos y Estructuras de Datos II
Laboratorio

22 de marzo de 2011

Este proyecto es una implementación en lenguaje C del proyecto de la materia Algorítmos y Estructuras de Datos I TAD diccionario sobre lista de asociaciones en el lenguaje Haskell (http://www.cs.famaf.unc.edu.ar/wiki/lib/exe/fetch.php?media=algo1:2010-2:proy3.pdf). Se recomienda su relectura antes de hacer el presente proyecto.

La implementación descripta a continuación está estructurada de forma jerárquica. Esto es, hay que implementar ciertos TAD's básicos (`data`, `key`, etc.) sobre los cuales se implementarán otros más complejos.



Todos los TAD's deberán ser implementados con la técnica de punteros a registros vista en el teórico del laboratorio.

De forma general, para hacer cada TAD se deberá tener en cuenta:

- Cada TAD se debera escribir en un par de archivos separados, un `.h` ("headers") y un `.c`.
- Para implementar correctamente los TAD's, cada .h debera exportar unicamente las funcionalidades del TAD que define, ocultando todos los aspectos que tienen que ver con su implementación.
- No definir las estructuras que implementan los TAD's en los archivos "headers" ya que allí solo va la parte pública del TAD y no los detalles de implementación.
- Para almacenar `Key` se utilizarán strings de tamaño fijo, `Data` **se utilizarán strings de tamaño arbitrario** (se debe usar memoria dinámica). La lista de asociaciones tiene tamaño fijo salvo que se quiera hacer el punto estrella.
- Los programas deben estar libres de "memory leaks" (utilizar el comando `valgrind`).
- Recordar que todos los `.h` deben tener su cerradura `ifndef`.
- Compilar todos los archivos y linkear el programa con las opciones, `-pedantic`, `-Wall` y `-std=c99`. Ver que no haya mensajes de error o advertencias.
- Utilizar el tipo *bool* de la librería `stdbool` (introducida en el estándar C *C99*).
- Para obtener nota B^+ se deben hacer todos los puntos estrella.
- No hacer los puntos estrella hasta que termine y pruebe el proyecto sin ellos.
- Se darán hechas las definiciones `typedef` de los tipos de los TAD's, pero se deberán hacer sus implementaciones (en los `.c` correspondientes).

A continuación se detallan los esquemas de los cuerpo de los `.h` de cada TAD a implementar. Se incluyen comentarios que deberán ser respetados en la implementación (en los `.c`).

1. Implementar el TAD Data. Basicamente hay implementar estas funciones en C:

```
typedef struct sData *  Data;

/* Construye un dato a partir de un string
   Se copia el contenido de s. */
Data
data_fromString (char *s);

/* Convierte un dato en un string en memoria dinámica.
   Se deberá liberarla cuando no se use. */
char *
data_toString (Data d);

/* Devuelve la cantidad de caracteres en el string almacenado.
```

```
   Esta función es de orden constante. */
int
data_lenght (Data d);

/* Devuelve un copia del dato
   (en nueva memoria). */
Data
data_clone (Data d);

/* Destructor.
   Destruye todo el contenido */
Data
data_destroy(Data d);
```

2. Implementar el TAD Key. Como este TAD almacenará información de tamaño acotado (a diferencia con `Data`) no hace falta que la estructura contenga un puntero. Las funciones a implementar son:

```
#include <stdbool.h>

typedef struct sKey * Key;

/* Devuelve el tamaño máximo
   (definido por KEY_MAX_LENGH en el \verb|.c|). */
int
key_maxLen (void);

/* Construye un clave desde un string.
   Se copia el contenido de s.
   Pre: strlen(s) <= que key_maxLen()
/* (devuelve error si esto no sucede).*/
Key
key_fromString (char *s);

/* Convierte una clave a string.
   El tamaño de s debe ser >= key_length()+1. */
void
key_toString (Key k, char *s);

/* Devuelve la longitud del string en k.
   Esta función es de orden constante */
int
key_length (Key k);

/* Pregunta si las claves son iguales. */
bool
key_eq (Key k1, Key k2);
```

```
/* Pregunta si la primer clave es menor que la segunda. */
bool
key_le (Key k1, Key k2);

/* Devuelve un copia
   (en nueva memoria). */
Key
key_clone (Key k);

/* Destructor
   destruye todo el contenido */
Key
key_destroy(Key k);
```

3. Como C carece de tuplas, hay que implementarlas. La tupla será un tipo contenedor por lo que su constructor no creará copias de sus coordenadas: se almacenarán sus referencias (punteros). La signatura debe ser:

```
#include "key.h"
#include "data.h"

typedef struct sTuple * Tuple;

/* Constructor.
   No copia k d: almacena las referencias (punteros).*/
Tuple
tuple_fromKeyData (Key k, Data d);

/* Devuelve una referencia (puntero) a la primer componente. */
Key
tuple_fst (Tuple t);

/* Devuelve una referencia a la segunda componente. */
Data
tuple_snd (Tuple t);

/* Devuelve un copia
   (copia Key Data y Tuple en nueva memoria). */
Tuple
tuple_clone (Tuple t);

/* Destructor.
   Destruye la tupla con su Key y Data. */
Tuple
tuple_destroy(Tuple t);
```

4. Implementar el TAD ListaAsoc (lista de asociaciones). Este TAD será un tipo contenedor por lo que su constructor no creará copias de sus elementos: se almacenarán sus referencias (punteros). Su signatura será:

```
#include <stdbool.h>
#include "key.h"
#include "data.h"

typedef struct sListaAsoc * ListaAsoc;

/* Constructor.
   Crea una lista vacía. */
ListaAsoc
la_empty (void);

/* Agrega una clave y un dato.
   Si la clave ya está el dato no se agrega. */
void
la_add (ListaAsoc l, Key k, Data d);

/* Busca un dato según la clave.
   Si no lo encuentra devuelve NULL. */
Data
la_search (ListaAsoc l, Key k);

/* Borra la clave y el valor asociado.
   Si la clave no está la lista no se modifica. */
void
la_del (ListaAsoc l, Key k);

/* Devuelve la cantidad de elementos.
   La función es de orden constante. */
int
la_length (ListaAsoc l);

/* Imprime en pantalla todos los pares (Key, Data).*/
void
la_pprint (ListaAsoc l);

/* Destructor.
   Destruye la lista y su contenido. */
ListaAsoc
la_destroy(ListaAsoc l);
```

Ejercicio $\star$ 1 *Hacer que el tamaño de la lista no sea fijo. Esto se puede hacer con la función* `realloc` *(ver man page). La idea es comenzar con un tamaño de* 1 *y si se supera este tamaño multiplicarlo por dos.*

5. Implementar el TAD Dict con la lista de asociaciones. La signatura debe ser:

```
#include <stdbool.h>

typedef char *Word;
typedef char *Def;

typedef struct sDict * Dict;

/* Constructor
   diccionario vacío.*/
Dict
dict_empty (void);

/* Agrega palabra y definicion.
   Pre: !dict_exists(d,w)
   (devuelve error si esto no sucede).*/
void
dict_add (Dict d, Word w, Def f);

/* Pregunta si la palabra está en el diccionario. */
bool
dict_exists (Dict d, Word w);

/* Devuelve la definición de una palabra
   almacenandola en memoria dinámica
   (acordarse de liberarla cuando no se use).
   Pre: dict_exists(d,w)
   (devuelve error si esto no sucede).*/
Def
dict_search (Dict d, Word w);

/* Remueve la palabra y la definicion del diccionario.
   Pre: dict_exists(d,w)
   (devuelve error si esto no sucede).*/
void
dict_del (Dict d, Word w);

/* Imprime todos los pares (clave, valor) en el diccionario. */
void
dict_pprint (Dict d);

/* Devuelve la cantidad de definiciones almacenadas.
   La función es de orden constante. */
int
dict_length (Dict d);
```

```
/* Destructor. */
Dict
dict_destroy (Dict d);

/* Maxima cantidad de caracteres del tipo Word */
int
word_maxlen(void);
```

Ejercicio $\star$ 2 *La búsqueda en el diccionario tiene como precondición que la palabra exista. Esto puede producir una perdida de eficiencia ya que cada ves que se quiera buscar una definición habrá que ver antes si la palabra existe. Pensar una forma de solucionar este problema.*

Ayuda: modificar la implementación de la lista de asociaciones para que incluya el último buscado (cache).

6. Desarrollar una interfaz similar a la que se utilizó para el proyecto de Algoritmos I sin la opción de grabar o leer el diccionario de disco. Solo debe utilizar las funciones y los tipos en `Dict`.