Proyecto 4
TAD Diccionario

Algoritmos y Estructuras de Datos II

2 de mayo de 2006

Implementar un TAD diccionario con arreglos utilizando los archivos dict.h, key.h y data.h en el apéndice A. Cada uno de estos da la especificación (descripción, PRE y POS) de los tres TAD que hay que implementar (diccionario, clave y dato respectivamente).

El diccionario debe poder ser guardado y leido desde disco. Para ello se brindan ya programadas los TAD's cinta de lectura y escritura de pares (ver en la página de la materia).

Para implementar el diccionario hay que tener en cuenta:

1. Leer detenidamente y entender las especificaciones de los TAD's que estan en los archivos dict.h, key.h y data.h.
2. Hacer lo mismo con las especificaciones de las cintas en los archivos cr.h, cw.h, parStr.h.
3. Se deben escribir tres archivos dict.c, key.c y data.c que implementen los TAD's respetando la especificación de las funciones dadas en sus respectivos .h.
4. NO ACCEDER A LA REPRESENTACIÓN INTERNA DE LOS TAD's fuera de cada archivo .c que lo implementa.
5. Hacer en otro archivo separado el bloque `main` donde el usuario pueda:
   - Cargar en memoria el diccionario desde disco.
   - Agregar y borrar palabras con sus definiciones.
   - Ver si una palabra existe en el diccionario.
   - Buscar la definición de una palabra.
   - Modificar la definición de una palabra.
   - Ver el tamaño del diccionario.
   - Guardar el diccionario en disco.
6. Los TAD's cinta de lectura y escritura solo deben ser utilizados en la implementación del diccionario (dict.c), NO en el bloque main.
7. Los TAD's cinta de lectura y escritura se brindan implementados en una librería estática. Esta librería se puede descargar de la página de la materia junto con un ejemplo de uso (copia un archivo en otro). Ver el ejemplo (archivo pru.c y Makefile) para poder utilizarla en el diccionario.

Para hacer todos los programas seguir la siguientes directivas generales:

- Ocultar toda la información sobre la implementacion de los TAD's al escribir sus implementaciones y en el bloque `main`.

- Poner los bloques `ifndef` en cada archivo .h.
- En los archivos .h agregar la especifición de cada función como se muestra en el apéndice A.
- Hacer lo mismo en los .c pero con las propiedades sobre los algoritmos concretos (o sea agregar invariantes y cotas).
- Hacer Makefile compilando todos los archivos con las opciones `-ansi`, `-pedantic` y `-Wall`. Ver que no haya mensajes de error o advertencias. Se puede usar el Makefile genérico que se brinda con la librería.
- No usar ninguna variable global.
- No dejar "memory leaks".

A. Headers de TAD's

A.1. dict.h

```
#ifndef DICT_H
#define DICT_H

#include "bool.h"
#include "key.h"
#include "data.h"

typedef struct sdict * dict;

dict
dict_empty(void);
/*
DESC: Constructor del tipo. Crea un diccionario vacio.
PRE: { d = NULL }
   d = dict_empty(f);
POS: {  Hay memoria  ==> d --> empty
      ! Hay memoria  ==> d == NULL }
*/

dict
dict_fromFile(const char * f);
/*
DESC: Constructor del tipo. Crea un diccionario a partir de un archivo.
      Toma el nombre del archivo.
PRE: { d = NULL }
   d = dict_fromFile(f);
POS: {  Hay memoria /\ ! Error de Archivo ==> d --> fromFile(f)
      ! Hay memoria \/   Error de Archivo ==> d == NULL }
*/

dict
dict_destroy(dict d);
/*
```

```
DESC: Destructor del tipo. d cambia.
PRE: { d --> D }
   d = dict_destroy(d);
POS: { Se destruye D /\ d == NULL }
*/

bool
dict_toFile(const char *f , const dict d);
/*
DESC: Guarda el diccionario en el archivo f.
PRE: { d --> D }
        b = dict_toFile(f, d);
POS: { ! Error de Archivo ==>  b /\ D guardado en archivo f
         Error de Archivo ==> !b }
*/

dict
dict_add(dict d, const key k, const data e);
/*
DESC: Agrega en el diccionario el dato d con la clave k.
      Si la clave ya existe no hace nada.
      Si agrega no utiliza k ni e (copia).
PRE: { d --> D /\ k --> K /\ e --> E /\ !exist(D,K)}
        d = dict_add(d, k, e);
POS: { d --> add(D,K,E) }
*/

bool
dict_exist(const dict d, const key k);
/*
DESC: Busca el dato con clave k si existe. Si no existe devuelve False.
PRE: { d --> D /\ k --> K }
        b = dict_exist(d, k);
POS: { (D =  add(D',K,E) /\ b ) \/
       (D != add(D',K,E) /\ !b ) }
*/

data
dict_search(const dict d, const key k);
/*
DESC: Busca y trae el dato con clave k si existe. Si no devuelve NULL.
      Crea copia del dato.
PRE: { d --> D /\ k --> K /\ e = NULL }
        e = dict_search(d, k);
POS: { (D =  add(D',K,E) /\ e --> E ) \/
       (D != add(D',K,E) /\ e = NULL ) }
*/

dict
dict_del(dict d, const key k);
```

```
/*
DESC: Borra el dato con clave k. Si no esta devuelve null.
      Destruye contenidos.
PRE: { d --> D /\ k --> K /\ e = NULL }
        d = dict_del(d, k);
POS: { ( D  = add(D',K,E) /\ d --> D') \/
       ( D != add(D',K,E) /\ d --> D }
*/

int
dict_length(dict d);
/*
DESC: Devuelve la cantidad de estradas en el diccionario.
PRE: { d --> D }
        l = dict_length(d);
POS: { l = length(D) }
*/

#endif /* DICT_H */
```

A.2. key.h

```
#ifndef KEY_H
#define KEY_H

#include "bool.h"

/* Maximo tamaño */
#define keyMaxLen 30

typedef struct skey * key;

key
key_empty(void);
/*
DESC: Construye un dato vacio.
PRE: { k = NULL }
   k = key_empty();
POS: {  Hay memoria ==> k --> empty
      ! Hay memoria ==> k == NULL }
*/

key
key_destroy(key k);
/*
DESC: Destructor del tipo. k cambia.
PRE: { k --> K }
   k = key_destroy(k);
POS: { Se destruye K /\ k == NULL }
*/
```

```
bool
key_copyStr(key k,const char *s);
/*
DESC: Transforma el string en key y lo copia k.
      Devuelve False si no hay memoria.
PRE: { k --> K /\ s --> S /\ len(S) <= keyMaxLen}
  b = key_copyStr(k,s);
POS: {  Hay memoria ==>  b /\ k --> fromStr(S)
      ! Hay memoria ==> !b /\ k --> K }
*/

char *
key_toStr(const key k);
/*
DESC: Devuelve un nuevo string (despues de usarlo hay que liberarlo) que
      representa la key.
PRE: { k --> K /\ s = NULL }
        s = key_toStr(k);
POS: {   Hay memoria ==> s --> toStr(K)
       ! Hay memoria ==> s == NULL }
*/

key
key_clone(const key k);
/*
DESC: Construye una copia.
PRE: { k1 --> K1 /\ k2 = NULL }
   k2 = key_clone(k1);
POS: {  Hay memoria ==> k1 --> K1 /\ k2 --> K1 /\ k1 != k2
      ! Hay memoria ==> k2 == NULL /\ k1 --> K1 }
*/

bool
key_eq(const key k1, const key k2);
/*
DESC: Igualdad. Se puede usar en expresiones.
PRE: { k1 --> K1 /\ k2 --> K2 }
   b = key_eq(k1,k2);
POS: { b == (K1 = K2) }
*/

bool
key_le(const key k1, const key k2);
/*
DESC: Menor. Se puede usar en expresiones.
PRE: { k1 --> K1 /\ k2 --> K2 }
   b = key_le(k1,k2);
POS: { b == (K1 < K2) }
*/
```

```
#endif /* KEY_H */
```

A.3. data.h

```
#ifndef DATA_H
#define DATA_H

#include "bool.h"

typedef struct sdata * data;

data
data_empty(void);
/*
DESC: Construye un dato vacio.
PRE: { True }
   d = data_empty();
POS: {  Hay memoria ==> d --> empty
      ! Hay memoria ==> d == NULL }
*/

data
data_destroy(data d);
/*
DESC: Destructor del tipo. d cambia.
PRE: { d --> D }
   d = data_destroy(d);
POS: { Se destruye D /\ d == NULL }
*/

bool
data_copyStr(data d,const char *s);
/*
DESC: Transforma el string en dato y lo copia d.
      Devuelve False si no hay memoria.
PRE: { d --> D /\ s --> S }
  b = data_copyStr(d,s);
POS: {  Hay memoria ==>  b /\ d --> fromStr(S)
      ! Hay memoria ==> !b /\ d --> D }
*/

char *
data_toStr(const data d);
/*
DESC: Devuelve un nuevo string (despues de usarlo hay que liberarlo) que
      representa al dato.
PRE: { d --> D  /\ s = NULL }
        s = data_toStr(d);
```

```
POS: {   Hay memoria ==> s --> toStr(S)
       ! Hay memoria ==> s == NULL  }
*/

data
data_clone(const data d);
/*
DESC: Construye una copia.
PRE: { d1 --> D1 /\ d2 = NULL }
   d2 = data_clone(d1);
POS: {  Hay memoria ==> d1 --> D1 /\ d2 --> D1 /\ d1 != d2
      ! Hay memoria ==> d2 == NULL /\ d1 --> D1 }
*/


#endif /* DATA_H */
```