# Proyecto 1
# TAD's en memoria dinámica

Algoritmos y Estructuras de Datos II
Laboratorio

4 de abril de 2008

Este proyecto es una reimplementación del proyecto de la materia Algorítmos y Estructuras de Datos I que trata sobre la implementación de un TAD diccionario sobre lista de asociaciones en el lenguaje C[1][2].

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

A diferencia del proyecto anterior, todos los TAD's deberán estar 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".
- Recordar que todos los `.h` deben tener su cerradura `ifndef`.
- Hacer el tipo `Bool` en una header separado.
- Para obtener nota $B^+$ se deben hacer todos los puntos estrella.
- Se darán hechas los definiciones `typedef` de los tipos de los TAD's, pero se deberán hacer sus implementaciones (en los `.c` correspondientes).

[1] Recordar que a su vez aquel proyecto respetaba la estructura de módulos de la versión del TAD diccionario en Haskell.

[2] Los proyectos de la materia Algorítmos y Estructuras de Datos I se encuentran en `http://hal.famaf.unc.edu.ar/~ayed1`.

A continuación se detallan los esquemas de los cuerpo de los `.h` de cada TAD a implementar.

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

```
/* ahora estamos usando punteros,
   asi que la definicion de la estructura
   no se encuentra en el header */
typedef struct sData *  Data;

/* crea un Dato a partir de un string */
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 el largo del string almacenado */
int
data_lenght (Data d);

/* devuelve un copia */
Data
data_clone (Data d);

/* Destructor */
Data
data_destroy(Data d);
```

2. Reimplementar el TAD Key. Basicamente hay implementar estas funciones en C:

```
typedef struct sKey * Key;

/* devuelve el largo máximo definido por KEY_MAX_LENGH */
int
key_maxLen (void);

/* construye un Key desde un string */
Key
key_fromString (char *s);

/* convierte un Key a string */
void
key_toString (Key k, char *s);

/* devuelve la longitud del string en k */
```

```
int
key_length (Key k);

/* pregunta si las claves son iguales */
Bool
key_eq (Key k1, Key k2);

/* pregunta si la primera clave es menor que la segunda */
Bool
key_le (Key k1, Key k2);

/* devuelve un copia */
Key
key_clone (Key k);

/* Destructor */
Key
key_destroy(Key d);
```

3. Como C carece de tuplas, hay que implementarlo. La signatura debe ser:

```
typedef struct sTuple * Tuple;

/* constructor */
Tuple
tuple_fromKeyData (Key k, Data s);

/* devuelve la primer componente */
Key
tuple_fst (Tuple t);

/* devuelve la segunda componente */
Data
tuple_snd (Tuple t);

/* devuelve un copia */
Tuple
tuple_clone (Tuple t);

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

4. Reimplementar el TAD ListaAsoc cuya signatura sea:

```
#define LIST_SIZE 100
```

```
typedef struct sListaAsoc * ListaAsoc;

/* constructor */
ListaAsoc
la_empty (void);

/* agrega una clave y un dato */
void
la_add (ListaAsoc l, Key k, Data d);

/* devuelve si existe la clave en la lista */
Bool
la_exists (ListaAsoc l, Key k);

/* busca un dato segun la clave
   debe llamarse solo si la_exists (l, k)
   si no lo encuentra devuelve NULL */
Data
la_search (ListaAsoc l, Key k);

/* borra la clave y el valor asociado */
void
la_del (ListaAsoc l, Key k);

/* devuelve la cantidad de elementos */
int
la_length (ListaAsoc l);

/* imprime todos los pares (Key, Data)*/
void
la_pprint (ListaAsoc l);

/* Destructor.
   Destruye la lista y su contenido */
ListAsoc
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* 10 *y si se supera este tamaño multiplicarlo por dos.*

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

```
typedef char *Word;
typedef char *Def;
```

```
typedef struct sDict * Dict;

/* constructor */
Dict
dict_empty (void);

/* agrega palabra y definicion */
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).
   Llamar solo si dict_exists (d, w)
   si no esta devuelve NULL */
Def
dict_search (Dict d, Word w);

/* saca la palabra y la definicion del diccionario */
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 */
int
dict_length (Dict d);

/* destructor */
Dict
dict_destroy (Dict d);
```

6. Desarrollar una interfaz similar a la que se entregó para el proyecto de Algoritmos I.