Geronimo Illescas – Padron 102071 – Francisco José Day – Padron 100513
Corrector : Martin Buchwald

```
/* ******************************************************************
 *                             TDA HASH
 * *****************************************************************/

#include <stdbool.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include "hash.h"
#include "lista.h"
#define TAM_INICIAL 1000
#define COEF_REDIM 2
#define UMBRAL_MAX 0.7
#define VALOR_MIN 4
/* ******************************************************************
 *                DEFINICION DE LOS TIPOS DE DATOS
 * *****************************************************************/

struct campo_hash{
      char* clave;
      void* valor;
}typedef campo_hash_t;

struct hash{
      lista_t** tabla;
      size_t tam; //(m que es la capacidad maxima de la estructura)
      size_t cant; //(n que es la cantidad de elementos que esta en el hash)
      hash_destruir_dato_t destruir;
};

struct hash_iter{
      const hash_t* hash;
      lista_iter_t* lista_iter;
      size_t pos;
      size_t iterados;
};

bool hash_redimensionar(hash_t* hash, size_t tam_nuevo);
lista_t** crear_tabla(size_t tam);

/* ******************************************************************
 *                DEFINICION DE FUNCIONES AUXILIARES
 * *****************************************************************/
/* Funcion hashing, recibe una cadena y el tamaños del hash. Duelve un
 * size_t.
 */
size_t funcion_hash(const char* s, size_t hash_tam){
      size_t hashvalue;
      for(hashvalue = 0; *s != '\0';s++)
            hashvalue = *s + 11 * hashvalue;
      return hashvalue % hash_tam;
}

/* Crea un hash nuevo, devuelve NULl si hubo algun problema
```

```
 */
hash_t* _hash_crear(hash_destruir_dato_t destruir_dato,size_t tam){
	hash_t* hash = malloc(sizeof(hash_t));
	if (!hash) return NULL;
	lista_t** tabla = crear_tabla(tam);
	if(!tabla){
		free(hash);
		return NULL;
	}
	hash->tam = tam;
	hash->cant = 0;
	hash->destruir = destruir_dato;
	hash->tabla = tabla;
	return hash;
}

/* Recibe una clave y un dato, y asocicia ambos parametros en un campo
 * La clave es copiada.
 */
campo_hash_t* crear_campo_hash(const char* clave, void* dato){
	campo_hash_t* campo_hash = malloc(sizeof(campo_hash_t));
	if(!campo_hash) return NULL;
	campo_hash->clave = malloc(sizeof(const char)* strlen(clave)+1);
	campo_hash->valor = dato;
	strcpy(campo_hash->clave, clave);
	return campo_hash;
}

/* Inicializa una tabla de hash con listas enlazadas (hash abierto)
 * En caso de que hubiera un problema al pedir memoria para la tabla o
 * una lista, devuelve NULL.
 */
lista_t** crear_tabla(size_t tam){
	lista_t** tabla = malloc(sizeof(lista_t*)* tam);
	if(!tabla) return NULL;

	for(int i = 0; i < tam; i++){
		tabla[i] = lista_crear();
		if(!tabla[i]){
			free(tabla);
			return NULL;
		}
	}
	return tabla;
}

/* Recibe un puntero a un struct hash y busca en el campo hash cuya
 * clave asignada sea la recibida por parametro, si tal campo no se
 * encontro devuelve NULL.
 * Pre: el hash fue creado
 * Post: Devuelve el campo si fue encontrado.
 */
campo_hash_t* buscar_campo_hash(const hash_t *hash, const char *clave){
	if(hash->cant == 0) return NULL;
	size_t pos = funcion_hash(clave, hash->tam);
```

```
	lista_iter_t* iter_lista = lista_iter_crear(hash->tabla[pos]);
	if(!iter_lista) return NULL;

	campo_hash_t* campo;
	while(!lista_iter_al_final(iter_lista)){
		campo = lista_iter_ver_actual(iter_lista);
		if(strcmp(campo->clave,clave) == 0){
			lista_iter_destruir(iter_lista);
			return campo;
		}
		lista_iter_avanzar(iter_lista);
	}
	lista_iter_destruir(iter_lista);
	return NULL;
}

/* Recibe una tabla de hash y su tamaño y se encarga en destruir todas
 * las listas interiores. Si destruir_dato es distinta de NULL se la
 * aplica sobre el valor de cada campo_hash.
 */
void destruir_tabla(lista_t** tabla, size_t tam, void destruir_dato(void*)){
	for(int i = 0; i < tam; i++){
		campo_hash_t* campo_hash = lista_borrar_primero(tabla[i]);
		while(campo_hash){
			if(destruir_dato){
				destruir_dato(campo_hash->valor);
			}
			free(campo_hash->clave);
			free(campo_hash);
			campo_hash = lista_borrar_primero(tabla[i]);
		}
		lista_destruir(tabla[i], free);
	}
	free(tabla);
}

/* Modifica el hash pasado por parametro redimensionandolo. Devuelve un
 * Segun si se modifico corretamento o no.
 */
bool hash_redimensionar(hash_t* hash, size_t tam_nuevo){
	hash_t* hash_redim = _hash_crear(hash->destruir,tam_nuevo);
	hash_iter_t* iter_hash = hash_iter_crear(hash);
	while(!hash_iter_al_final(iter_hash)){
		campo_hash_t* campo_aux = lista_iter_ver_actual(iter_hash->lista_iter);
		if(!hash_guardar(hash_redim, campo_aux->clave, campo_aux->valor)){
			hash_iter_destruir(iter_hash);
			hash_destruir(hash_redim);
			return false;
		}
		hash_iter_avanzar(iter_hash);
	}
	hash_iter_destruir(iter_hash);
	destruir_tabla(hash->tabla, hash->tam, NULL);
	hash->cant = hash_redim->cant;
	hash->destruir = hash_redim->destruir;
	hash->tabla = hash_redim->tabla;
```

```
	hash->tam = tam_nuevo;
	free(hash_redim);
	return true;
}



/* ******************************************************************
 *                    PRIMITIVAS DEL HASH
 * *****************************************************************/

/* Crea el hash
 */

hash_t* hash_crear(hash_destruir_dato_t destruir_dato){
	return _hash_crear(destruir_dato,TAM_INICIAL);
}

/* Recibe un puntero a un campo hash y elimina el dato, la clave y el
 * campo- Si recibe un puntero nulo no hace nada.
 */
void destruir_campo_hash(hash_t *hash, campo_hash_t* campo){
	if(!campo) return;
	if(hash->destruir)
		hash->destruir(campo->valor);
	free(campo->clave);
	free(campo);
}

/* Guarda un elemento en el hash, si la clave ya se encuentra en la
 * estructura, la reemplaza. De no poder guardarlo devuelve false.
 * Pre: La estructura hash fue inicializada
 * Post: Se almacenó el par (clave, dato)
 */
bool hash_guardar(hash_t *hash, const char *clave, void *dato) {
	if((hash->cant/hash->tam) >= UMBRAL_MAX)
		hash_redimensionar(hash, hash->tam * COEF_REDIM);

	if(hash_pertenece(hash, clave)){
		campo_hash_t* campo = buscar_campo_hash(hash, clave);
		if(hash->destruir)
			hash->destruir(campo->valor);
		campo->valor = dato;
		return true;
	}

	size_t indice = funcion_hash(clave, hash->tam);
	campo_hash_t* campo = crear_campo_hash(clave, dato);
	if(!campo || !lista_insertar_ultimo(hash->tabla[indice], campo)){
		destruir_campo_hash(hash,campo);
		return false;
	}

	hash->cant++;
	return true;
}
```

Geronimo Illescas – Padron 102071 – Francisco José Day – Padron 100513
Corrector : Martin Buchwald

```
/* Borra un elemento del hash y devuelve el dato asociado.  Devuelve
 * NULL si el dato no estaba.
 * Pre: La estructura hash fue inicializada
 * Post: El elemento fue borrado de la estructura y se lo devuelve
 * en el caso de que estuviera guardado. Queda en manos del usuario
 * la memoria de ese dato guardado.
 */
void* hash_borrar(hash_t *hash, const char *clave){
      if(hash->cant == 0) return NULL;
      if((hash->cant* VALOR_MIN)<= hash->tam && hash->cant * COEF_REDIM >= TAM_INICIAL)
         hash_redimensionar(hash, hash->cant*COEF_REDIM);
      size_t indice = funcion_hash(clave, hash->tam);
      if(lista_esta_vacia(hash->tabla[indice])) return NULL;
      lista_iter_t* iter = lista_iter_crear(hash->tabla[indice]);
      if(!iter) return NULL;
      campo_hash_t* registro;
      while((registro = lista_iter_ver_actual(iter))){
            if(strcmp(registro->clave, clave) == 0){
                  campo_hash_t* aux = lista_iter_borrar(iter);
                  void* dato = aux->valor;
                  lista_iter_destruir(iter);
                  free(aux->clave);
                  free(aux);
                  hash->cant--;
                  return dato;
            }
            lista_iter_avanzar(iter);
      }
      free(iter);
      return NULL;
}

/* Obtiene el valor de un elemento del hash, si la clave no se encuentra
 * devuelve NULL.
 * Pre: La estructura hash fue inicializada
 */
void* hash_obtener(const hash_t *hash, const char *clave){
      campo_hash_t* campo = buscar_campo_hash(hash, clave);
      if(!campo) return NULL;
      return campo->valor;
}

/* Determina si clave pertenece o no al hash.
 * Pre: La estructura hash fue inicializada
 */
bool hash_pertenece(const hash_t* hash, const char* clave){
      return buscar_campo_hash(hash, clave)!=NULL;
}

/* Devuelve la cantidad de elementos del hash.
 * Pre: La estructura hash fue inicializada
 */
size_t hash_cantidad(const hash_t* hash){
      return hash->cant;
}
```

```
/* Destruye la estructura liberando la memoria pedida y llamando a la función
 * destruir para cada par (clave, dato).
 * Pre: La estructura hash fue inicializada
 * Post: La estructura hash fue destruida
 */
void hash_destruir(hash_t *hash){
	destruir_tabla(hash->tabla, hash->tam, hash->destruir);
	free(hash);
}

/* Iterador del hash */

/* Crea iterador. Asigna el iterador al primer elemento de la tabla de
 * hash cuya lista no sea vacia. Si todas las listas estan vacias hace
 * referencia a NULL.
 * Pre: el hash fue creado.
 */
hash_iter_t* hash_iter_crear(const hash_t *hash){
	hash_iter_t* iter =  malloc(sizeof(hash_iter_t));
	if(!iter) return NULL;
	iter->hash = hash;
	iter->iterados = 0;
	iter->lista_iter = NULL;
	if(hash->cant == 0){
		iter->pos = 0;
		return iter;
	}

	size_t i = 0;
	while(lista_esta_vacia(hash->tabla[i]))
		i++;
	iter->lista_iter = lista_iter_crear(hash->tabla[i]);
	if(!iter->lista_iter){
		free(iter);
		return NULL;
	}
	iter->pos = i;
	return iter;
}

/* Avanza iterador sobre un mismo hash.
 * Pre: el iterador fue creado.
 */
bool hash_iter_avanzar(hash_iter_t *iter){
	if (hash_iter_al_final(iter)) return false;
	if (lista_iter_avanzar(iter->lista_iter) && !lista_iter_al_final(iter-
>lista_iter)) {
		iter->iterados++;
		return true;
	}
	lista_iter_destruir(iter->lista_iter);
	iter->pos++;
	iter->iterados++;
	size_t i = iter->pos;
	while(i < iter->hash->tam){
```

```
            if(!lista_esta_vacia(iter->hash->tabla[i])){
                  iter->lista_iter = lista_iter_crear(iter->hash->tabla[i]);
                  if(!iter->lista_iter) return false;
                  iter->pos = i;
                  return true;
            }
            i++;
      }
      iter->lista_iter = NULL;
      iter->pos = iter->hash->tam - 1;
      return false;
}

/* Devuelve clave actual, esa clave no se puede modificar ni liberar.
 */
const char* hash_iter_ver_actual(const hash_iter_t *iter){
      if(!iter || hash_iter_al_final(iter))
            return NULL;
      campo_hash_t* actual = lista_iter_ver_actual(iter->lista_iter);
      return actual->clave;
}

/* Comprueba si existen nodos siguientes en la lista actual o si
 * existen elemetos de la tabla hash no vacios. De esta manera si hay
 * algun elemento siguiente es posible iterar o no.
 */
bool hash_iter_al_final(const hash_iter_t *iter){
      return iter->iterados == iter->hash->cant;
}

/* Destruye iterador.
 * Pre: el iterador fue creado.
 */
void hash_iter_destruir(hash_iter_t* iter){
      if(iter->lista_iter){
            lista_iter_destruir(iter->lista_iter);
      }
      free(iter);
}
```

Geronimo Illescas – Padron 102071 – Francisco José Day – Padron 100513
Corrector : Martin Buchwald

```
/* ******************************************************************
 *                             HASH.H
 * *****************************************************************/
#ifndef HASH_H
#define HASH_H

#include <stdbool.h>
#include <stddef.h>

// Los structs deben llamarse "hash" y "hash_iter".
struct hash;
struct hash_iter;

typedef struct hash hash_t;
typedef struct hash_iter hash_iter_t;

// tipo de función para destruir dato
typedef void (*hash_destruir_dato_t)(void *);

/* Crea el hash
 */
hash_t *hash_crear(hash_destruir_dato_t destruir_dato);

/* Guarda un elemento en el hash, si la clave ya se encuentra en la
 * estructura, la reemplaza. De no poder guardarlo devuelve false.
 * Pre: La estructura hash fue inicializada
 * Post: Se almacenó el par (clave, dato)
 */
bool hash_guardar(hash_t *hash, const char *clave, void *dato);

/* Borra un elemento del hash y devuelve el dato asociado.  Devuelve
 * NULL si el dato no estaba.
 * Pre: La estructura hash fue inicializada
 * Post: El elemento fue borrado de la estructura y se lo devolvió,
 * en el caso de que estuviera guardado.
 */
void *hash_borrar(hash_t *hash, const char *clave);

/* Obtiene el valor de un elemento del hash, si la clave no se encuentra
 * devuelve NULL.
 * Pre: La estructura hash fue inicializada
 */
void *hash_obtener(const hash_t *hash, const char *clave);

/* Determina si clave pertenece o no al hash.
 * Pre: La estructura hash fue inicializada
 */
bool hash_pertenece(const hash_t *hash, const char *clave);

/* Devuelve la cantidad de elementos del hash.
 * Pre: La estructura hash fue inicializada
```

```
 */
size_t hash_cantidad(const hash_t *hash);

/* Destruye la estructura liberando la memoria pedida y llamando a la función
 * destruir para cada par (clave, dato).
 * Pre: La estructura hash fue inicializada
 * Post: La estructura hash fue destruida
 */
void hash_destruir(hash_t *hash);

/* Iterador del hash */

// Crea iterador
hash_iter_t *hash_iter_crear(const hash_t *hash);

// Avanza iterador
bool hash_iter_avanzar(hash_iter_t *iter);

// Devuelve clave actual, esa clave no se puede modificar ni liberar.
const char *hash_iter_ver_actual(const hash_iter_t *iter);

// Comprueba si terminó la iteración
bool hash_iter_al_final(const hash_iter_t *iter);

// Destruye iterador
void hash_iter_destruir(hash_iter_t* iter);

#endif // HASH_H
```