```c
/* ******************************************************************
 *      NOMBRE Y APELLIDO : LUCIANO NICOLAS MONTES PADRON: 102536
 *      NOMBRE Y APELLIDO : AGUSTIN LOPEZ NUÑEZ    PADRON: 101826
 * *****************************************************************/


#include <stdlib.h>
#include <stdbool.h>
#include "lista.h"
#include "hash.h"
#include <string.h>
#include <stdio.h>


/* ******************************************************************
 *                           CONSTANTES
 * *****************************************************************/

#define TAM_INICIAL 94
#define FACTOR_CARGA_MAX 3 //300% cargado
#define FACTOR_CARGA_MIN 0.20 //20% cargado
#define ENCONTRADO 0
#define DIVIDIR 2
#define DUPLICAR 2

/* ******************************************************************
 *                           ESTRUCTURAS
 * *****************************************************************/

struct hash{
    size_t cant;
    lista_t **tabla;
    size_t tam;
    hash_destruir_dato_t f_destruir;
};

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

typedef struct elemento_hash{
    char *clave;
    void *dato;
} elemento_hash_t;



/* ******************************************************************
 *                      FUNCION DE HASH(DJB2)
 * *****************************************************************/

size_t f_hash(const char *str, size_t tam) {
  size_t hash = 5381;
  int c;

  while ((c = *str++))
    hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

```

```c
  return hash % tam;
}

/* ******************************************************************
 *                      FUNCIONES AUXILIARES
 * *****************************************************************/

elemento_hash_t *elemento_hash_crear(void *dato, char *clave){
    elemento_hash_t *nodo = malloc(sizeof(elemento_hash_t));

    if(!nodo) return NULL;

    nodo->dato = dato;
    nodo->clave = clave;

    return nodo;
}

// Retorna el elemento buscado, NULL en caso de no encontrarse. Si se recibio un iterador de lista por parametro
// lo devuelve apuntando a ese elemento, o al final de la lista, si este no se encuentra. Si no se recibio iteador, se crea y destruye uno nuevo.
elemento_hash_t *buscar_elemento_hash(const hash_t *hash, const char *clave, lista_iter_t* p_lista_iter){
    size_t pos_hash = f_hash(clave, hash->tam);
    elemento_hash_t *elemento_hash = NULL;
    lista_iter_t *lista_iter = p_lista_iter;
    bool encontro = false;

    if(lista_esta_vacia(hash->tabla[pos_hash])) return NULL;

    if(!p_lista_iter){
        lista_iter = lista_iter_crear(hash->tabla[pos_hash]);
    }
    if(!lista_iter) return NULL;

    while((!lista_iter_al_final(lista_iter)) && (!encontro)){
        elemento_hash = lista_iter_ver_actual(lista_iter);
        (strcmp(elemento_hash->clave, clave) == ENCONTRADO) ? (encontro = true) : (lista_iter_avanzar(lista_iter));
    }

    if(!p_lista_iter) lista_iter_destruir(lista_iter);

    if(!encontro) return NULL;

    return elemento_hash;
}
char *copiar_clave(const char* clave){
    char *clave_copia = malloc(strlen(clave) + 1);
    if(!clave_copia) return false;

    strcpy(clave_copia, clave);
    return clave_copia;
}

bool pos_prox_lista_ocupada(const hash_t *hash, size_t *pos_actual){

    if(*(pos_actual)+1 == hash->tam) return false;

```

```c
    *(pos_actual)+=1;

    while((lista_esta_vacia(hash->tabla[*pos_actual])) && (*pos_actual+1 < hash->tam)){
        *(pos_actual)+=1;
    }

    if(lista_esta_vacia(hash->tabla[*pos_actual])) return false;

    return true;
}


bool reasignar_tabla(hash_t *hash, size_t nuevo_tam, lista_t **tabla_nueva){
    lista_t **tabla_anterior = hash->tabla;
    size_t tam_anterior = hash->tam;

    hash->tabla = tabla_nueva;
    hash->tam = nuevo_tam;

    size_t pos_actual = 0;
    bool guarda_ok = true;
    elemento_hash_t *elemento_hash;

    while((pos_actual < tam_anterior) && (guarda_ok)){
        if(tabla_anterior[pos_actual]){
            while((!lista_esta_vacia(tabla_anterior[pos_actual])) && (guarda_ok)){
                elemento_hash = lista_borrar_primero(tabla_anterior[pos_actual]);
                guarda_ok = lista_insertar_ultimo(tabla_nueva[f_hash(elemento_hash->clave, hash->tam)], elemento_hash);
            }
        }
        lista_destruir(tabla_anterior[pos_actual],NULL);
        pos_actual++;
    }

    free(tabla_anterior);
    return guarda_ok;
}


bool hash_redimensionar(hash_t *hash, size_t nuevo_tam){
    lista_t **tabla_nueva = malloc(sizeof(lista_t*)* nuevo_tam);

    if(!tabla_nueva) return false;

    for (size_t i = 0; i < nuevo_tam; i++) {
    tabla_nueva[i] = lista_crear();
    }

    return reasignar_tabla(hash, nuevo_tam, tabla_nueva);
}


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

```c
/* Crea el hash
 */
hash_t *hash_crear(hash_destruir_dato_t destruir_dato){
    hash_t *hash = malloc(sizeof(hash_t));
    lista_t **tabla = malloc(sizeof(lista_t*) * TAM_INICIAL);

    if((!hash) || (!tabla)){
    free(hash);
    free(tabla);
    return NULL;
  }

    hash->cant = 0;
    hash->tam = TAM_INICIAL;
    hash->tabla = tabla;
    hash->f_destruir = destruir_dato;

    for (size_t i = 0; i < hash->tam; i++) {
    hash->tabla[i] = lista_crear();
    }

    return hash;
}


/* Guarda un elemento en el hash, si la clave ya se encuentra en#include <stddef.h> 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){
    float factor_carga = (float)hash->cant / (float)hash->tam;

  if(factor_carga > FACTOR_CARGA_MAX){
        if(!hash_redimensionar(hash, hash->tam * DUPLICAR)) return false;
    }

    char *clave_copia = copiar_clave(clave);
    elemento_hash_t *elemento_hash = buscar_elemento_hash(hash, clave_copia, NULL);

    if(elemento_hash != NULL){
        if(hash->f_destruir) hash->f_destruir(elemento_hash->dato);
        free(clave_copia);
        elemento_hash->dato = dato;
        return true;
    }

  elemento_hash = elemento_hash_crear(dato, clave_copia);

    if(!elemento_hash || !(lista_insertar_ultimo(hash->tabla[f_hash(clave, hash->tam)],elemento_hash))){
    free(clave_copia);
        free(elemento_hash);
    return false;
  }

  hash->cant++;

```

```c
    return true;
}


/* Borra un elemento del hash y devuelve el dato asociado.  Devuelve(float)
 * 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){
    float factor_carga = (float)hash->cant / (float)hash->tam;

    if((factor_carga < FACTOR_CARGA_MIN) && (hash->tam/DIVIDIR > TAM_INICIAL)){
        hash_redimensionar(hash, hash->tam / DIVIDIR);
    }

    size_t pos_hash = f_hash(clave, hash->tam);

    if(lista_esta_vacia(hash->tabla[pos_hash])) return NULL;

    lista_iter_t *lista_iter = lista_iter_crear(hash->tabla[pos_hash]);
    elemento_hash_t *elemento_hash = buscar_elemento_hash(hash, clave, lista_iter);

    if(!lista_iter || !elemento_hash){
        lista_iter_destruir(lista_iter);
        return NULL;
    }

    void *dato = elemento_hash->dato;

    free(elemento_hash->clave);
    free(elemento_hash);
    lista_iter_borrar(lista_iter);
    lista_iter_destruir(lista_iter);
    hash->cant--;

    return dato;
}


/* 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){
    elemento_hash_t *elemento_hash = buscar_elemento_hash(hash, clave, NULL);

    if(!elemento_hash) return NULL;

    return elemento_hash->dato;
}


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

```c
    return buscar_elemento_hash(hash,clave, NULL) != 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){
    elemento_hash_t *nodo;

    for(size_t i = 0; i < hash->tam; i++){
        while(!lista_esta_vacia(hash->tabla[i])){
            nodo = lista_borrar_primero(hash->tabla[i]);
            free(nodo->clave);
            if(hash->f_destruir) hash->f_destruir(nodo->dato);
            free(nodo);
        }
        lista_destruir(hash->tabla[i], NULL);
    }

    free(hash->tabla);
    free(hash);
}



/* ******************************************************************
 *                      PRIMITIVAS DEL ITERADOR
 * *****************************************************************/

// Crea iterador
hash_iter_t *hash_iter_crear(const hash_t *hash){
    hash_iter_t *hash_iter = malloc(sizeof(hash_iter_t));
    size_t pos_actual = 0;

    if(!hash_iter) return NULL;

    pos_prox_lista_ocupada(hash, &pos_actual);

    hash_iter->pos = pos_actual;
    hash_iter->lista_iter = lista_iter_crear(hash->tabla[hash_iter->pos]);

    if(!hash_iter->lista_iter){
        hash_iter_destruir(hash_iter);
        return NULL;
    }

    hash_iter->hash = hash;

```

```
347     return hash_iter;
348 }
349 
350 
351 // Avanza iterador
352 bool hash_iter_avanzar(hash_iter_t *iter){
353 
354     if(hash_iter_al_final(iter)) return false;
355 
356     lista_iter_avanzar(iter->lista_iter);
357 
358     if(lista_iter_al_final(iter->lista_iter)){
359         if(!pos_prox_lista_ocupada(iter->hash, &iter->pos)) return false;
360 
361         lista_iter_destruir(iter->lista_iter);
362         iter->lista_iter = lista_iter_crear(iter->hash->tabla[iter->pos]);
363 
364         if(!(iter->lista_iter)) return false;
365     }
366 
367     return true;
368 }
369 
370 
371 // Devuelve clave actual, esa clave no se puede modificar ni liberar.
372 const char *hash_iter_ver_actual(const hash_iter_t *iter){
373 
374     if(hash_iter_al_final(iter)) return NULL;
375 
376     elemento_hash_t* elemento_hash = lista_iter_ver_actual(iter->lista_iter);
377     return elemento_hash->clave;
378 }
379 
380 
381 // Comprueba si terminó la iteración
382 bool hash_iter_al_final(const hash_iter_t *iter){
383     bool final_tabla = false;
384     size_t pos_tabla = iter->pos;
385 
386     if(!pos_prox_lista_ocupada(iter->hash, &pos_tabla)) final_tabla = true;
387 
388     return(final_tabla && lista_iter_al_final(iter->lista_iter));
389 }
390 
391 
392 // Destruye iterador
393 void hash_iter_destruir(hash_iter_t* iter){
394 
395     lista_iter_destruir(iter->lista_iter);
396     free(iter);
397 }
398 
399 
```