**Apellidos: Cardozo - Allende Cantidad de hojas: 5**

**Padrones: 98750 - 98559 Corrector Jorge Collinet**

#include "hash.h"

#include <stdlib.h>

#include <stdio.h>

#include <stdbool.h>

#include <string.h>

#define FACTOR\_CARGA 70

#define REDIM\_SUP 2

#define REDIM\_INF 0.5

#define MIN\_CAPAC 0.25

#define TAM\_INICIAL 31

typedef enum {BORRADO,VACIO,OCUPADO} estado\_t;

typedef struct campo\_hash campo\_hash\_t;

struct campo\_hash{

char\* clave;

void\* dato;

estado\_t estado;

};

struct hash{

size\_t cantidad;

size\_t tamanio;

campo\_hash\_t\* tabla;

hash\_destruir\_dato\_t destructor;

};

// funcion de hash la cual al introducir un clave me devuelve una posicion de la tabla.

size\_t func\_hash(const char \*word, size\_t hashTableSize){

int hashAddress = 0;

for (int counter = 0; word[counter]!='\0'; counter++){

hashAddress = word[counter] + (hashAddress << 6) + (hashAddress << 16) - hashAddress;

}

return (size\_t)(hashAddress%hashTableSize);

}

//funciona auxiliar la cual me permite buscar la posicion en la que se encuentra una clave existente

//en la tabla de hash. Si encontro la posicion devuelvo true.

//En caso de que corte porque encontre una posicion con estado VACIO o BORRADO si esp\_guardar es true.

bool buscando\_pos\_en\_tabla( const hash\_t\* hash, size\_t\* pos\_hash, const char \*clave,bool esp\_guardar){

// Parto de que si no es OCUPADO no me interesa, es decir no me interesa BORRADO.

bool clave\_encontrada = false;

while( hash -> tabla[\*pos\_hash].estado != VACIO && !clave\_encontrada ) {

if( hash->tabla[\*pos\_hash].estado == OCUPADO && strcmp( hash -> tabla[\*pos\_hash].clave ,clave ) == 0 ){

clave\_encontrada = true;

}

else{

if ( esp\_guardar && hash->tabla[\*pos\_hash].estado == BORRADO ) break;

(\*pos\_hash)++;

if( \*pos\_hash == hash -> tamanio ) \*pos\_hash = 0;

}

}

return clave\_encontrada;

}

/\* 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 guardar\_elem\_en\_tabla(hash\_t \*hash, const char \*clave, void \*dato){

size\_t h = func\_hash(clave,hash -> tamanio);

if( hash\_pertenece( hash,clave ) ){

buscando\_pos\_en\_tabla(hash,&h,clave,false);

if( hash->destructor != NULL ){

hash->destructor(hash->tabla[h].dato);

}

hash->tabla[h].dato = dato;

return true;

}

buscando\_pos\_en\_tabla(hash,&h,clave,true);

if( hash->tabla[h].estado == BORRADO){

hash ->tabla[h].clave = realloc( hash ->tabla[h].clave,(strlen(clave)+1)\*sizeof(char) );

}

else{

// pido strlen(clave) + 1 porque tengo que contemplar el largo de la cadena y el '\0' de al final.

hash->tabla[h].clave = malloc( ( strlen(clave) + 1 )\*sizeof(char) );

}

if( hash ->tabla[h].clave == NULL ) return false;

strcpy( hash->tabla[h].clave , clave );

hash->tabla[h].dato = dato;

hash->cantidad++;

hash->tabla[h].estado = OCUPADO;

return true;

}

//Inicializa los estados de cada campo en VACIO.

void inicializar\_tabla(campo\_hash\_t \*tabla, size\_t tam){

for(int i = 0; i < tam; i++){

tabla[i].estado = VACIO;

}

}

//Destruye el contenido de la tabla pero según el caso.

//Si el ultimo parametro es false solo destruyos las claves y si es true destruyo el par (clave,dato).

void destruir\_contenido\_tabla(campo\_hash\_t\* tabla,size\_t tamanio,hash\_destruir\_dato\_t destructor,bool dest\_todo){

for (int pos\_hash = 0; pos\_hash < tamanio; ++pos\_hash){

if( tabla[pos\_hash].estado == VACIO ) continue;

if( dest\_todo && destructor && tabla[pos\_hash].estado == OCUPADO ) destructor( tabla[pos\_hash].dato );

free( tabla[pos\_hash].clave );

}

}

//Redimensiona la tabla segun el caso,agrandar como achicar, de manera que se pueda seguir usando.

//Devuelve un True si la redimension se pudo llevar a cabo y en caso contrario False.

bool redimensionar\_tabla(hash\_t\* hash,size\_t tam\_nuevo){

campo\_hash\_t\* tabla\_nueva = malloc( tam\_nuevo \* sizeof(campo\_hash\_t) );

if( tabla\_nueva == NULL ) return false;

campo\_hash\_t\* tabla\_vieja = hash -> tabla;

hash -> tabla = tabla\_nueva;

size\_t tam\_viejo = hash -> tamanio;

hash -> tamanio = tam\_nuevo;

inicializar\_tabla( hash-> tabla, hash -> tamanio);

//tengo que hashear cada par (clave,dato) de nuevo.

for (int indice\_hash = 0; indice\_hash < tam\_viejo; indice\_hash++){

if( tabla\_vieja[indice\_hash].estado == OCUPADO){

guardar\_elem\_en\_tabla( hash , tabla\_vieja[indice\_hash].clave , tabla\_vieja[indice\_hash].dato);

hash->cantidad--;

}

}

destruir\_contenido\_tabla(tabla\_vieja,tam\_viejo,hash->destructor,false);

free(tabla\_vieja);

return true;

}

/\* Crea el hash

\*/

hash\_t \*hash\_crear(hash\_destruir\_dato\_t destruir\_dato){

hash\_t\* tu\_hash = malloc( sizeof(hash\_t) );

if( !tu\_hash ) return NULL;

tu\_hash -> tamanio = TAM\_INICIAL;

tu\_hash -> cantidad = 0;

tu\_hash -> destructor = destruir\_dato;

tu\_hash -> tabla = (campo\_hash\_t\*)malloc( sizeof(campo\_hash\_t)\* tu\_hash->tamanio );

if( !tu\_hash -> tabla ){

free( tu\_hash );

return NULL;

}

inicializar\_tabla( tu\_hash->tabla , tu\_hash->tamanio );

return tu\_hash;

}

/\* Determina si clave pertenece o no al hash.

\* Pre: La estructura hash fue inicializada

\*/

bool hash\_pertenece(const hash\_t \*hash, const char \*clave){

size\_t pos\_hash = func\_hash( clave, hash -> tamanio );

return buscando\_pos\_en\_tabla(hash,&pos\_hash,clave,false);

}

/\* 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 (!guardar\_elem\_en\_tabla(hash,clave,dato)) return false;

if( ( (float)hash->cantidad / (float)hash->tamanio ) \* 100 >= FACTOR\_CARGA ){

if( !redimensionar\_tabla( hash , (size\_t)( (float)hash->tamanio \* REDIM\_SUP ) ) ) return false;

}

return true;

}

/\* 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){

if( !hash\_pertenece(hash,clave) || clave == NULL ) return NULL;

// ahora se que la clave existe dentro de mi tabla de hash.

size\_t pos\_hash = func\_hash(clave,hash -> tamanio);

buscando\_pos\_en\_tabla(hash,&pos\_hash,clave,false);

void\* dato\_a\_borrar = hash -> tabla[pos\_hash].dato;

hash -> tabla[pos\_hash].estado = BORRADO;

hash -> cantidad--;

if( hash -> cantidad < ( (float)hash->tamanio \* MIN\_CAPAC ) && hash -> tamanio != TAM\_INICIAL ){

redimensionar\_tabla( hash,(size\_t)( (float)hash-> tamanio \* REDIM\_INF ) );

}

return dato\_a\_borrar;

}

/\* 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){

if( !hash\_pertenece(hash,clave) ) return NULL;

size\_t pos\_hash = func\_hash(clave,hash -> tamanio);

buscando\_pos\_en\_tabla(hash,&pos\_hash,clave,false);

return hash -> tabla[pos\_hash].dato;

}

/\* Devuelve la cantidad de elementos del hash.

\* Pre: La estructura hash fue inicializada

\*/

size\_t hash\_cantidad(const hash\_t \*hash){

return hash -> cantidad;

}

/\* 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\_contenido\_tabla( hash->tabla , hash->tamanio , hash->destructor, true);

free( hash -> tabla );

free( hash );

}

/\* ITERADOR DEL HASH \*/

struct hash\_iter{

hash\_t\* hash;

size\_t indice;

};

// Crea iterador

hash\_iter\_t \*hash\_iter\_crear(const hash\_t \*hash){

hash\_iter\_t\* iter\_hash = malloc( sizeof(hash\_iter\_t) );

if( iter\_hash == NULL ) return NULL;

iter\_hash -> hash = (hash\_t\*)hash;

iter\_hash -> indice = 0;

while( iter\_hash -> hash->tabla[iter\_hash ->indice].estado != OCUPADO && !hash\_iter\_al\_final(iter\_hash) ){

iter\_hash -> indice++;

}

return iter\_hash;

}

// Avanza iterador

bool hash\_iter\_avanzar(hash\_iter\_t \*iter){

if( hash\_iter\_al\_final(iter) ) return false;

iter->indice++;

while( !hash\_iter\_al\_final(iter) && (iter -> hash -> tabla[ iter -> indice ].estado != OCUPADO) ){

iter->indice++;

}

if( hash\_iter\_al\_final(iter) ) return false;

return true;

}

// Devuelve clave actual, esa clave no se puede modificar ni liberar.

const char \*hash\_iter\_ver\_actual(const hash\_iter\_t \*iter){

if( hash\_iter\_al\_final(iter) ) return NULL;

return iter -> hash -> tabla[ iter -> indice ].clave;

}

// Comprueba si terminó la iteración

bool hash\_iter\_al\_final(const hash\_iter\_t \*iter){

if ( iter -> indice == iter->hash->tamanio || iter-> hash->cantidad == 0 ) return true;

return false;

}

// Destruye iterador

void hash\_iter\_destruir(hash\_iter\_t\* iter){

free(iter);

}