/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 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;

}

/\* 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);

}

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 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