#include "abb.h"

#include "math.h"

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include "pila.h"

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

\* DEFINICION DE LOS TIPOS DE DATOS

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

typedef struct nodo {

struct nodo\* izq;

struct nodo\* der;

char \*clave;

void \*dato;

}abb\_nodo\_t;

typedef struct abb {

abb\_nodo\_t\* raiz;

abb\_comparar\_clave\_t cmp;

abb\_destruir\_dato\_t destruir\_dato;

size\_t cant\_nodos;

}abb\_t;

typedef struct abb\_iter{

abb\_nodo\_t\* actual;

pila\_t\* pila;

}abb\_iter\_t;

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

\* DEFINICION DE FUNCIONES AUXILIARES

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

/\* Crea un nodo para un arbol binario de busqueda. Si fallo el pedido

\* de memoria devuleve NULL.

\*/

abb\_nodo\_t\* crear\_nodo\_abb(const char\* clave,void\* dato){

if(!clave) return NULL;

abb\_nodo\_t\* nodo\_abb = malloc(sizeof(abb\_nodo\_t));

if (!nodo\_abb) return NULL;

nodo\_abb->izq = NULL;

nodo\_abb->der = NULL;

nodo\_abb->dato = dato;

nodo\_abb->clave = malloc(sizeof(char)\*strlen(clave)+1);

if(! nodo\_abb->clave){

free(nodo\_abb);

return NULL;

}

strcpy(nodo\_abb->clave,clave);

return nodo\_abb;

}

void\* nodo\_destruir(abb\_nodo\_t\* nodo){

if(!nodo)

return NULL;

void\* dato = nodo->dato;

free(nodo->clave);

free(nodo);

return dato;

}

/\* Recibe un nodo de arbol binario de busqueda, un clave y una fucnion

\* de comparacion. Busca en los subarboles del nodo el nodo identificado

\* con la clave pasada por parametro. Si no se encuntra devuelve NULL.

\* Pre: cmp fue definida.

\*/

abb\_nodo\_t\* abb\_nodo\_buscar(abb\_nodo\_t\* nodo,const char\* clave, abb\_comparar\_clave\_t cmp){

if(!nodo) return NULL;

int i = cmp(nodo->clave, clave);

if(i > 0)

return abb\_nodo\_buscar(nodo->izq,clave,cmp);

if(i < 0)

return abb\_nodo\_buscar(nodo->der,clave,cmp);

return nodo;

}

/\* Recibe un nodo de arbol binario de busqueda, un clave, un nodo

\* anterior y una fucnion de comparacion. Busca en los subarboles del

\* nodo el nodo identificado con la clave pasada por parametro y devuelve

\* el nodo padre.Si no se encuntra devuelve NULL.

\* Pre: cmp fue definida.

\*/

abb\_nodo\_t\* abb\_nodo\_buscar\_padre(abb\_nodo\_t\* nodo,abb\_nodo\_t\* padre,const char\* clave, abb\_comparar\_clave\_t cmp){

if(!nodo) return NULL;

int i = cmp(nodo->clave, clave);

if(i > 0)

return abb\_nodo\_buscar\_padre(nodo->izq,nodo,clave,cmp);

if(i < 0)

return abb\_nodo\_buscar\_padre(nodo->der,nodo,clave,cmp);

return padre;

}

/\* Busca el nodo cuya clave sea la minima a partir del nodo pasado.

\* Pre: El nodo pasado por parametro existe

\* Post: Devuelve el nodo de minima clave

\*/

abb\_nodo\_t\* abb\_nodo\_buscar\_minimo(abb\_nodo\_t\* nodo){

if(!nodo) return NULL;

if(!nodo->izq)

return nodo;

return abb\_nodo\_buscar\_minimo(nodo->izq);

}

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

\* PRIMITIVAS DEL ABB

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

/\* Recibe funciones para comparar los datos entre si y para destruirlos

\* Devuelve un puntero a un arbol binario vacio (raiz nula)

\* Pre: cmp y destruir\_dato fueron previamente definidas.

\*/

abb\_t\* abb\_crear(abb\_comparar\_clave\_t cmp, abb\_destruir\_dato\_t destruir\_dato){

abb\_t \* arbol = malloc(sizeof(abb\_t));

if(!arbol) return NULL;

arbol->raiz = NULL;

arbol->destruir\_dato = destruir\_dato;

arbol->cmp = cmp;

arbol->cant\_nodos = 0;

return arbol;

}

/\* Reciben un valor y una clave asociada al valor y lo guarda en el arbol.

\* Devuleve un boolenao si la operacion fue definida.

\* Pre: el arbol fue previamente creado.

\*/

bool \_abb\_guardar(abb\_nodo\_t \*nodo, abb\_nodo\_t\* padre,abb\_t \*arbol, const char \*clave, void \*dato){

if(!nodo){

nodo = crear\_nodo\_abb(clave,dato);

if(!nodo)

return false;

if(arbol->cmp(padre->clave,clave)<0)

padre->der = nodo;

else

padre->izq = nodo;

arbol->cant\_nodos++;

return true;

}

int i = arbol->cmp(nodo->clave, clave);

//es positivo si la clave a guardar es menor

//es negativo si la clave a guardar es mayor

if(i > 0)

return \_abb\_guardar(nodo->izq,nodo,arbol,clave,dato);

if(i < 0)

return \_abb\_guardar(nodo->der,nodo,arbol,clave,dato);

if(arbol->destruir\_dato)

arbol->destruir\_dato(nodo->dato);

nodo->dato = dato;

return true;

}

bool abb\_guardar(abb\_t \*arbol, const char \*clave, void \*dato){

if(!arbol->raiz){

arbol->raiz = crear\_nodo\_abb(clave,dato);

if(!arbol->raiz)

return false;

arbol->cant\_nodos++;

return true;

}

if(!\_abb\_guardar(arbol->raiz,NULL,arbol,clave,dato))

return false;

return true;

}

void\* abb\_borrar\_sin\_hijos(abb\_t\* arbol, abb\_nodo\_t\* nodo, abb\_nodo\_t\* padre){

if(!padre) {

arbol->raiz = NULL;

arbol->cant\_nodos--;

return nodo\_destruir(nodo);

}

int i = arbol->cmp(padre->clave, nodo->clave);

if(i > 0) padre->izq = NULL;

if(i < 0) padre->der = NULL;

arbol->cant\_nodos--;

return nodo\_destruir(nodo);

}

void\* abb\_borrar\_hijo\_unico(abb\_t\* arbol, abb\_nodo\_t\* nodo, abb\_nodo\_t\* padre){

abb\_nodo\_t\* aux;

if(nodo->izq && !nodo->der){//Tiene hijo izquierdo

if(!padre)

arbol->raiz = nodo->izq;

else

aux = nodo->izq;

}

if(nodo->der && !nodo->izq){//Tiene hijo derecho

if (!padre)

arbol->raiz = nodo->der;

else

aux = nodo->der;

}

if(padre){

int i = arbol->cmp(padre->clave,nodo->clave);

if (i > 0) padre->izq = aux;

if (i < 0) padre->der = aux;

}

arbol->cant\_nodos--;

return nodo\_destruir(nodo);

}

void\* abb\_borrar\_dos\_hijos(abb\_t\* arbol, abb\_nodo\_t\* nodo, abb\_nodo\_t\* padre){

abb\_nodo\_t\* reemplazo = abb\_nodo\_buscar\_minimo(nodo->der);

abb\_nodo\_t\* padre\_reemplazo = abb\_nodo\_buscar\_padre(nodo->der,nodo,reemplazo->clave,arbol->cmp);

if(arbol->cmp(padre\_reemplazo->clave, nodo->clave) == 0)

reemplazo->izq = nodo->izq;

else{

padre\_reemplazo->izq = reemplazo->der;

reemplazo->der = nodo->der;

reemplazo->izq = nodo->izq;

}

if(!padre)

arbol->raiz = reemplazo;

else{

int i = arbol->cmp(padre->clave,nodo->clave);

if(i > 0)

padre->izq = reemplazo;

if(i < 0)

padre->der = reemplazo;

}

arbol->cant\_nodos--;

return nodo\_destruir(nodo);;

}

/\* Recibe una clave y se encarga de eliminar el dato asociado y lo

\* devuelve. Si la clave no pertenece a ningun dato del arbol, o este

\* esta vacio se devuelve NULL.

\* Pre: el arbol fue creado.

\*/

void\* abb\_borrar(abb\_t \*arbol, const char \*clave){

if(!abb\_pertenece(arbol, clave)) return NULL;

abb\_nodo\_t\* nodo = abb\_nodo\_buscar(arbol->raiz,clave,arbol->cmp);

abb\_nodo\_t\* padre = abb\_nodo\_buscar\_padre(arbol->raiz,NULL,clave,arbol->cmp);

if(!nodo->der && !nodo->izq)

return abb\_borrar\_sin\_hijos(arbol, nodo, padre);

if(!nodo->izq || !nodo->der)

return abb\_borrar\_hijo\_unico(arbol, nodo, padre);

if(nodo->izq && nodo->der)

return abb\_borrar\_dos\_hijos(arbol, nodo, padre);

return NULL;

}

/\* Recibe un arbol y una clave asociada a un valor, se buca en el arbol

\* el valor asociado, si se lo encuentra se lo devuelve. Caso contrario

\* devuelve NULL.

\* Pre: el arbol fue creado.

\*/

void \*abb\_obtener(const abb\_t \*arbol, const char \*clave){

abb\_nodo\_t\* nodo = abb\_nodo\_buscar(arbol->raiz,clave,arbol->cmp);

if(!nodo)

return NULL;

return nodo->dato;

}

/\* Recibe un arbol y una clave asociada a un valor, se buca en el arbol

\* el valor asociado, si se lo encuentra se devulve true. Caso contrario

\* devuelve false.

\* Pre: el arbol fue creado.

\*/

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

if(!arbol->raiz)

return false;

return abb\_nodo\_buscar(arbol->raiz, clave, arbol->cmp) != NULL;

}

/\* Devuelve la cantida de claves que hay en el arbol.

\* Pre: el arbol fue creado.

\*/

size\_t abb\_cantidad(abb\_t \*arbol){

if(!arbol)

return 0;

return arbol->cant\_nodos;

}

void \_abb\_destruir(abb\_nodo\_t\* nodo, abb\_destruir\_dato\_t destruir\_dato){

if(nodo->izq)

\_abb\_destruir(nodo->izq, destruir\_dato);

if(nodo->der)

\_abb\_destruir(nodo->der, destruir\_dato);

void\* dato = nodo\_destruir(nodo);

if(destruir\_dato)

destruir\_dato(dato);

}

void abb\_destruir(abb\_t \*arbol){

if(arbol->raiz){

\_abb\_destruir(arbol->raiz, arbol->destruir\_dato);

}

free(arbol);

}

/\* Itera de manera inorder sobre el arbol aplicandole la funcion visitar

\* a cada clave y dato asociado que se encuentre.

\* Pre: el arbol fue creado.

\*/

bool \_abb\_in\_order(abb\_nodo\_t\* nodo, bool visitar(const char\*, void\*, void\*), void\* extra){

if(!nodo) return true;

if(\_abb\_in\_order(nodo->izq,visitar,extra)){

if(visitar(nodo->clave,nodo->dato,extra))

return \_abb\_in\_order(nodo->der,visitar,extra);

}

return false;

}

void abb\_in\_order(abb\_t\* arbol, bool visitar(const char\*, void\*, void\*), void\* extra){

if(!arbol->raiz) return;

\_abb\_in\_order(arbol->raiz,visitar,extra);

}

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

\* PRIMITIVAS DEL ITERADOR

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

bool apilar\_hijos\_izquierdos(abb\_iter\_t\* iter, abb\_nodo\_t\* nodo){

while(nodo){

if(!pila\_apilar(iter->pila, nodo))

return false;

nodo = nodo->izq;

}

return true;

}

abb\_iter\_t\* abb\_iter\_in\_crear(const abb\_t\* arbol){

abb\_iter\_t\* iter = malloc(sizeof(abb\_iter\_t));

if(!iter)

return NULL;

pila\_t\* pila = pila\_crear();

if(!pila){

free(iter);

return NULL;

}

iter->pila = pila;

if(!apilar\_hijos\_izquierdos(iter, arbol->raiz)){

pila\_destruir(iter->pila);

abb\_iter\_in\_destruir(iter);

return NULL;

}

iter->actual = pila\_ver\_tope(iter->pila);

return iter;

}

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

if(iter->actual == NULL)

return NULL;

return iter->actual->clave;

}

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

return pila\_esta\_vacia(iter->pila);

}

bool abb\_iter\_in\_avanzar(abb\_iter\_t\* iter){

if(abb\_iter\_in\_al\_final(iter))

return false;

abb\_nodo\_t\* nodo = pila\_desapilar(iter->pila);

if(nodo->der)

apilar\_hijos\_izquierdos(iter, nodo->der);

iter->actual = pila\_ver\_tope(iter->pila);

return true;

}

void abb\_iter\_in\_destruir(abb\_iter\_t\* iter){

pila\_destruir(iter->pila);

free(iter);

}

#include "testing.h"

#include "abb.h"

#include <stdio.h>

#include <string.h>

#include <stddef.h>

#include <stdlib.h>

#include <time.h>

void random\_inicializar(){

unsigned int seed = (unsigned int)time(NULL);

srand (seed);

}

int nuestro\_random(int lim){

return rand()%lim;

}

// Funciones de swapeo

void swap\_char(char\*\* x, char\*\* y){

char\* aux=\*x;

\*x=\*y;

\*y=aux;

}

void swap\_int(int\*\* x, int\*\* y){

int\* aux=\*x;

\*x=\*y;

\*y=aux;

}

void vector\_desordenar(char\* claves[], int\* valores[], int largo){

random\_inicializar();

int i;

int rnd;

// es importante que el par clave-valor se mantenga siempre igual

for (i=0; i<largo;i++){

rnd=nuestro\_random(largo);

swap\_char(&claves[i], &claves[rnd]);

swap\_int(&valores[i], &valores[rnd]);

}

}

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

\* PRUEBAS PARA ABB

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

// Realiza pruebas sobre la implementación de la abb del alumno.

//

// Las pruebas deben emplazarse en el archivo ‘pruebas\_alumno.c’, y

// solamente pueden emplear la interfaz pública tal y como aparece en abb.h

// (esto es, las pruebas no pueden acceder a los miembros del struct abb).

//

// Para la implementación de las pruebas se debe emplear la función

// print\_test(), como se ha visto en TPs anteriores.

void abb\_vacio(){

printf("\nIncio pruebas abb vacias\n");

abb\_t\* abb = abb\_crear(strcmp, NULL);

print\_test("Guardar cadena nula ",!abb\_guardar(abb,NULL,NULL));

print\_test(("Clave 'perro' no pertenece: "), !abb\_pertenece(abb, "Perro"));

print\_test("Cantidad es 0 ",abb\_cantidad(abb)==0);

print\_test("Borrar clave es NULL ",!abb\_borrar(abb,NULL));

abb\_destruir(abb);

}

void abb\_simple(){

printf("\nSimples\n");

abb\_t\* abb = abb\_crear(strcmp, NULL);

char \*clave1 = "perro", \*valor1 = "guau";

print\_test("Guardar clave1 ",abb\_guardar(abb,clave1,valor1));

print\_test("Cantidad es 1 ",abb\_cantidad(abb)==1);

print\_test("Pertence clave1? ",abb\_pertenece(abb,clave1));

print\_test("Obtener ",strcmp(abb\_obtener(abb,clave1),valor1)==0);

print\_test("Eliminamos ",strcmp(abb\_borrar(abb,clave1),valor1)==0);

print\_test("Cantidad es 0 ",abb\_cantidad(abb)==0);

abb\_destruir(abb);

}

void abb\_multiples(){

printf("\nInicio pruebas multiples\n");

abb\_t\* abb = abb\_crear(strcmp,free);

char \*clave1 = "perro", \*valor1 = "guau";

char \*clave2 = "gato", \*valor2 = "miau";

char \*clave3 = "vaca", \*valor3 = "mu";

print\_test("Guardar clave1 ",abb\_guardar(abb,clave1,valor1));

print\_test("Cantidad es 1 ",abb\_cantidad(abb)==1);

print\_test("Pertence clave1? ",abb\_pertenece(abb,clave1));

print\_test("Obtener ",strcmp(abb\_obtener(abb,clave1),valor1)==0);

print\_test("Guardar clave2 ",abb\_guardar(abb,clave2,valor2));

print\_test("Cantidad es 2 ",abb\_cantidad(abb)==2);

print\_test("Pertence clave2? ",abb\_pertenece(abb,clave2));

print\_test("Obtener ",strcmp(abb\_obtener(abb,clave2),valor2)==0);

print\_test("Guardar clave3 ",abb\_guardar(abb,clave3,valor3));

print\_test("Cantidad es 3 ",abb\_cantidad(abb)==3);

print\_test("Pertence clave3? ",abb\_pertenece(abb,clave3));

print\_test("Obtener ",strcmp(abb\_obtener(abb,clave3),valor3)==0);

print\_test("Eliminamos clave1 ",strcmp(abb\_borrar(abb,clave1),valor1)==0);

print\_test("Eliminamos clave2 ",strcmp(abb\_borrar(abb,clave2),valor2)==0);

print\_test("Eliminamos clave3 ",strcmp(abb\_borrar(abb,clave3),valor3)==0);

print\_test("Cantidad es 0 ",abb\_cantidad(abb)==0);

abb\_destruir(abb);

}

void abb\_volumen(){

printf("Prueba de arbol a volumen\n");

abb\_t\* arbol=abb\_crear(strcmp,NULL);

int largo=1000;

char\* claves[largo];

int\* valores[largo];

int i;

for (i = 0; i < largo; i++) {

claves[i] = malloc(10\*sizeof(char));

valores[i] = malloc(sizeof(int));

sprintf(claves[i], "%08d", i);

\*valores[i] = i;

}

vector\_desordenar(claves,valores,largo);

bool ok=true;

i=0;

while (i<largo && ok){

ok=abb\_guardar(arbol, claves[i], valores[i]);

i++;

}

print\_test("Puedo agregar 10000 valores", ok);

print\_test("La cantidad de elementos del arbol es 10000", abb\_cantidad(arbol)==largo);

i=0;

ok=true;

while(ok && i<largo){

ok = abb\_pertenece(arbol, claves[i]);

ok = abb\_obtener(arbol, claves[i]) == valores[i];

i++;

}

print\_test("Los elementos estan bien guardados, y pertenecen",ok);

print\_test("Hay 10000 elementos en el arbol", abb\_cantidad(arbol)==largo);

/\* Verifica que borre y devuelva los valores correctos \*/

i=0; ok=true;

while (i < largo && ok) {

ok = abb\_borrar(arbol, claves[i]) == valores[i];

i++;

}

print\_test("Los elementos al borrarlos dieron todos bien", ok);

print\_test("Ahora el arbol esta vacio", abb\_cantidad(arbol)==0);

abb\_destruir(arbol);

arbol = abb\_crear(strcmp,free);

/\* Inserta 'largo' parejas en el hash \*/

ok = true;

i=0;

while(i < largo && ok) {

ok = abb\_guardar(arbol, claves[i], valores[i]);

i++;

}

/\* Libera las cadenas \*/

for (i = 0; i < largo; i++) {

free(claves[i]);

}

/\* Destruye el arbol - debería liberar los enteros \*/

abb\_destruir(arbol);

}

void pruebas\_iter\_arbol\_vacio(){

printf("\nInicio pruebas con iterador en arbol vacio: \n");

abb\_t\* arbol = abb\_crear(strcmp, NULL);

abb\_iter\_t\* iter = abb\_iter\_in\_crear(arbol);

print\_test("Iter creado: ",iter!=NULL);

print\_test("Actual es NULL: ", abb\_iter\_in\_ver\_actual(iter) == NULL);

print\_test("Avanza en arbol vacio: ", !abb\_iter\_in\_avanzar(iter));

print\_test("Iter esta al final: ", abb\_iter\_in\_al\_final(iter));

abb\_iter\_in\_destruir(iter);

abb\_destruir(arbol);

print\_test("Arbol e iter destruidos: ", true);

}

void pruebas\_iter\_elementos(){

printf("\nInicio pruebas del iter con elementos\n");

abb\_t\* arbol = abb\_crear(strcmp, NULL);

int valor1 = 1;

int valor2 = 2;

int valor3 = 3;

print\_test("Guardo un elemento: ", abb\_guardar(arbol, "Perro", &valor1));

print\_test("Guardo un elemento: ", abb\_guardar(arbol, "Gato", &valor2));

print\_test("Guardo un elemento: ", abb\_guardar(arbol, "Vaca", &valor3));

print\_test("Elemento(perro) pertenece al arbol: ", abb\_pertenece(arbol, "Perro"));

print\_test("Elemento(gato) pertenece al arbol: ", abb\_pertenece(arbol, "Gato"));

print\_test("Elemento(vaca) pertenece al arbol: ", abb\_pertenece(arbol, "Vaca"));

abb\_iter\_t\* iter = abb\_iter\_in\_crear(arbol);

print\_test("Iter no esta al final: ", !abb\_iter\_in\_al\_final(iter));

print\_test("Elemento actual es 'Gato: ", strcmp(abb\_iter\_in\_ver\_actual(iter), "Gato") == 0);

print\_test("Avance OK: ", abb\_iter\_in\_avanzar(iter));

print\_test("Iter no esta al final: ", !abb\_iter\_in\_al\_final(iter));

print\_test("Elemento actual es 'Perro': ", strcmp(abb\_iter\_in\_ver\_actual(iter), "Perro") == 0);

print\_test("Avance OK: ", abb\_iter\_in\_avanzar(iter));

print\_test("Iter no esta al final: ", !abb\_iter\_in\_al\_final(iter));

print\_test("Elemento actual es 'Vaca': ", strcmp(abb\_iter\_in\_ver\_actual(iter), "Vaca") == 0);

print\_test("Avance OK, ", abb\_iter\_in\_avanzar(iter));

print\_test("Iter al final ", abb\_iter\_in\_al\_final(iter));

print\_test("Actual es NULL: ", abb\_iter\_in\_ver\_actual(iter) == NULL);

print\_test("No puedo avanzar: ", !abb\_iter\_in\_avanzar(iter));

abb\_iter\_in\_destruir(iter);

abb\_destruir(arbol);

print\_test("Arbol e iter destruidos: ", true);

}

void pruebas\_abb\_iterar\_volumen(size\_t largo){

printf("\nInicio pruebas del iter volumen\n");

abb\_t\* abb = abb\_crear(strcmp, NULL);

const size\_t largo\_clave = 10;

char (\*claves)[largo\_clave] = malloc(largo \* largo\_clave);

size\_t valores[largo];

/\* Inserta 'largo' parejas en el abb \*/

bool ok = true;

for (unsigned i = 0; i < largo; i++) {

sprintf(claves[i], "%08d", i);

valores[i] = i;

ok = abb\_guardar(abb, claves[i], &valores[i]);

if (!ok) break;

}

// Prueba de iteración sobre las claves almacenadas.

abb\_iter\_t\* iter = abb\_iter\_in\_crear(abb);

print\_test("Prueba abb iterador esta al final, es false", !abb\_iter\_in\_al\_final(iter));

ok = true;

unsigned i;

const char \*clave;

size\_t \*valor;

for (i = 0; i < largo; i++) {

if ( abb\_iter\_in\_al\_final(iter) ) {

ok = false;

break;

}

clave = abb\_iter\_in\_ver\_actual(iter);

if ( clave == NULL ) {

ok = false;

break;

}

valor = abb\_obtener(abb, clave);

if ( valor == NULL ) {

ok = false;

break;

}

\*valor = largo;

abb\_iter\_in\_avanzar(iter);

}

print\_test("Prueba abb iteración en volumen", ok);

print\_test("Prueba abb iteración en volumen, recorrio todo el largo", i == largo);

print\_test("Prueba abb iterador esta al final, es true", abb\_iter\_in\_al\_final(iter));

ok = true;

for (i = 0; i < largo; i++) {

if ( valores[i] != largo ) {

ok = false;

break;

}

}

print\_test("Prueba abb iteración en volumen, se cambiaron todo los elementos", ok);

free(claves);

abb\_iter\_in\_destruir(iter);

abb\_destruir(abb);

}

bool funcion(const char\* clave, void\* dato,void\* extra){

printf("La clave es %s y el dato %d\n",clave,\*(int\*)dato);

return true;

}

void iter\_interno(){

printf("\nInicio pruebas del iter interno\n");

abb\_t\* abb = abb\_crear(strcmp, NULL);

char\* claves[10]={"5","4","6","7","1","2","3","9","8","0"};

int datos[10]={5,4,6,7,1,2,3,9,8,0};

abb\_in\_order(abb,funcion,NULL);

for(int i=0;i<10;i++){

abb\_guardar(abb,claves[i],&datos[i]);

}

abb\_in\_order(abb,funcion,NULL);

abb\_destruir(abb);

}

void pruebas\_abb\_alumno(void){

abb\_vacio();

abb\_simple();

abb\_multiples();

abb\_volumen();

iter\_interno();

pruebas\_iter\_arbol\_vacio();

pruebas\_iter\_elementos();

pruebas\_abb\_iterar\_volumen(5000);

printf("Se termino correctamente el programa\n");

}