Geronimo Illescas – Padron 102071 – Francisco José Day – Padron 100513
Corrector : Martin Buchwald G43

```
#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;
}
```

Geronimo Illescas – Padron 102071 – Francisco José Day – Padron 100513
Corrector : Martin Buchwald G43

```
/* 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);
}
```

Geronimo Illescas – Padron 102071 – Francisco José Day – Padron 100513
Corrector : Martin Buchwald G43

```
#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");
}
```