75.41 Algoritmos y Programación II Curso 4
TDA Arbol

Binario de Búsqueda

29 de mayo de 2020

1. Enunciado

Se pide implementar un Arbol Binario de Búsqueda. Para ello se brindan las firmas de las funciones públicas a implementar y se deja a criterio del alumno la creación de las funciones privadas del TDA para el correcto funcionamiento del Arbol cumpliendo con las buenas prácticas de programación.

Adicionalmente se pide la implementación de un iterador interno para la estructura.

2. abb.h

```c
#ifndef __ARBOL_BINARIO_DE_BUSQUEDA_H__
#define __ARBOL_BINARIO_DE_BUSQUEDA_H__

#define ABB_RECORRER_INORDEN   0
#define ABB_RECORRER_PREORDEN  1
#define ABB_RECORRER_POSTORDEN 2

#include <stdbool.h>
#include <stdlib.h>

/*
 * Comparador de elementos. Recibe dos elementos del arbol y devuelve
 * 0 en caso de ser iguales, 1 si el primer elemento es mayor al
 * segundo o -1 si el primer elemento es menor al segundo.
 */
typedef int (*abb_comparador)(void*, void*);

/*
 * Destructor de elementos. Cada vez que un elemento deja el arbol
 * (arbol_borrar o arbol_destruir) se invoca al destructor pasandole
 * el elemento.
 */
typedef void (*abb_liberar_elemento)(void*);


typedef struct nodo_abb {
        void* elemento;
        struct nodo_abb* izquierda;
        struct nodo_abb* derecha;
} nodo_abb_t;

typedef struct abb{
        nodo_abb_t* nodo_raiz;
        abb_comparador comparador;
        abb_liberar_elemento destructor;
} abb_t;

/*
 * Crea el arbol y reserva la memoria necesaria de la estructura.
 * Comparador se utiliza para comparar dos elementos.
 * Destructor es invocado sobre cada elemento que sale del arbol,
 * puede ser NULL indicando que no se debe utilizar un destructor.
 *
 * Devuelve un puntero al arbol creado o NULL en caso de error.
 */
abb_t* arbol_crear(abb_comparador comparador, abb_liberar_elemento destructor);

```

```c
/*
 * Inserta un elemento en el arbol.
 * Devuelve 0 si pudo insertar o -1 si no pudo.
 * El arbol admite elementos con valores repetidos.
 */
 int arbol_insertar(abb_t* arbol, void* elemento);

/*
 * Busca en el arbol un elemento igual al provisto (utilizando la
 * funcion de comparación) y si lo encuentra lo quita del arbol.
 * Adicionalmente, si encuentra el elemento, invoca el destructor con
 * dicho elemento.
 * Devuelve 0 si pudo eliminar el elemento o -1 en caso contrario.
 */
int arbol_borrar(abb_t* arbol, void* elemento);

/*
 * Busca en el arbol un elemento igual al provisto (utilizando la
 * funcion de comparación).
 *
 * Devuelve el elemento encontrado o NULL si no lo encuentra.
 */
void* arbol_buscar(abb_t* arbol, void* elemento);

/*
 * Devuelve el elemento almacenado como raiz o NULL si el árbol está
 * vacío o no existe.
 */
void* arbol_raiz(abb_t* arbol);

/*
 * Determina si el árbol está vacío.
 * Devuelve true si está vacío o el arbol es NULL, false si el árbol tiene elementos.
 */
bool arbol_vacio(abb_t* arbol);

/*
 * Llena el array del tamaño dado con los elementos de arbol
 * en secuencia inorden.
 * Devuelve la cantidad de elementos del array que pudo llenar (si el
 * espacio en el array no alcanza para almacenar todos los elementos,
 * llena hasta donde puede y devuelve la cantidad de elementos que
 * pudo poner).
 */
int arbol_recorrido_inorden(abb_t* arbol, void** array, int tamanio_array);

/*
 * Llena el array del tamaño dado con los elementos de arbol
 * en secuencia preorden.
 * Devuelve la cantidad de elementos del array que pudo llenar (si el
 * espacio en el array no alcanza para almacenar todos los elementos,
 * llena hasta donde puede y devuelve la cantidad de elementos que
 * pudo poner).
 */
int arbol_recorrido_preorden(abb_t* arbol, void** array, int tamanio_array);

/*
 * Llena el array del tamaño dado con los elementos de arbol
 * en secuencia postorden.
 * Devuelve la cantidad de elementos del array que pudo llenar (si el
 * espacio en el array no alcanza para almacenar todos los elementos,
 * llena hasta donde puede y devuelve la cantidad de elementos que
 * pudo poner).
 */
int arbol_recorrido_postorden(abb_t* arbol, void** array, int tamanio_array);

/*
 * Destruye el arbol liberando la memoria reservada por el mismo.
 * Adicionalmente invoca el destructor con cada elemento presente en
 * el arbol.
 */
void arbol_destruir(abb_t* arbol);

/*
 * Iterador interno. Recorre el arbol e invoca la funcion con cada
 * elemento del mismo. El puntero 'extra' se pasa como segundo
```

```
 * parámetro a la función. Si la función devuelve true, se finaliza el
 * recorrido aun si quedan elementos por recorrer. Si devuelve false
 * se sigue recorriendo mientras queden elementos.
 * El recorrido se realiza de acuerdo al recorrido solicitado.  Los
 * recorridos válidos son: ABB_RECORRER_INORDEN, ABB_RECORRER_PREORDEN
 * y ABB_RECORRER_POSTORDEN.
*/
void abb_con_cada_elemento(abb_t* arbol, int recorrido, bool (*funcion)(void*, void*), void* extra);

#endif /* __ARBOL_BINARIO_DE_BUSQUEDA_H__ */
```

3. Compilación y Ejecución

El TDA entregado deberá compilar y pasar las pruebas dispuestas por la cátedra sin errores, adicionalmente estas pruebas deberán ser ejecutadas sin pérdida de memoria.

Compilación:

```
gcc *.c -o abb -g -std=c99 -Wall -Wconversion -Wtype-limits -pedantic -Werror -O0
```

Ejecución:

```
valgrind --leak-check=full --track-origins=yes --show-reachable=yes ./abb
```

4. Minipruebas

Se les brindará un lote de minipruebas, las cuales recomendamos fuertemente sean ampliadas ya que no son exhaustivas y no prueban todos los casos borde, solo son un ejemplo de como agregar, eliminar, obtener y buscar elementos dentro del árbol y qué debería verse en la terminal en el **caso feliz**.

Minipruebas:

```
#include "abb.h"
#include <stdio.h>

typedef struct cosa {
	int clave;
	char contenido[10];
} cosa_t;

cosa_t* crear_cosa(int clave){
	cosa_t* c = (cosa_t*)malloc(sizeof(cosa_t));
	if(c)
		c->clave = clave;
	return c;
}

void destruir_cosa(cosa_t* c){
	if(c)
		free(c);
}

int comparar_cosas(void* elemento1, void* elemento2){
	if(!elemento1 || !elemento2)
		return 0;

	if(((cosa_t*)elemento1)->clave>((cosa_t*)elemento2)->clave)
		return 1;

	if(((cosa_t*)elemento1)->clave<((cosa_t*)elemento2)->clave)
		return -1;

	return 0;
}

void destructor_de_cosas(void* elemento){
	if(!elemento)
		return;

	destruir_cosa((cosa_t*)elemento);
}

bool mostrar_elemento(void* elemento, void* extra){
	extra=extra; //para que no se queje el compilador, gracias -Werror -Wall
```

```
43
44          if(elemento)
45                  printf("%i ", ((cosa_t*)elemento)->clave);
46
47          return false;
48  }
49
50  bool mostrar_hasta_5(void* elemento, void* extra){
51          extra=extra; //para que no se queje el compilador, gracias -Werror -Wall
52
53          if(elemento){
54                  printf("%i ", ((cosa_t*)elemento)->clave);
55                  if(((cosa_t*)elemento)->clave == 5)
56                          return true;
57          }
58
59          return false;
60  }
61
62  bool mostrar_acumulado(void* elemento, void* extra){
63          if(elemento && extra){
64                  *(int*)extra += ((cosa_t*)elemento)->clave;
65                  printf("%i ", *(int*)extra);
66          }
67
68          return false;
69  }
70
71
72  int main(){
73          abb_t* arbol = arbol_crear(comparar_cosas, destructor_de_cosas);
74
75          cosa_t* c1= crear_cosa(1);
76          cosa_t* c2= crear_cosa(2);
77          cosa_t* c3= crear_cosa(3);
78          cosa_t* c4= crear_cosa(4);
79          cosa_t* c5= crear_cosa(5);
80          cosa_t* c6= crear_cosa(6);
81          cosa_t* c7= crear_cosa(7);
82          cosa_t* auxiliar = crear_cosa(0);
83
84          arbol_insertar(arbol, c4);
85          arbol_insertar(arbol, c2);
86          arbol_insertar(arbol, c6);
87          arbol_insertar(arbol, c1);
88          arbol_insertar(arbol, c3);
89          arbol_insertar(arbol, c5);
90          arbol_insertar(arbol, c7);
91
92          printf("El nodo raiz deberia ser 4: %s\n", ((cosa_t*)arbol_raiz(arbol))->clave==4?"SI":"NO")
    ;
93
94          auxiliar->clave = 5;
95          printf("Busco el elemento 5: %s\n", ((cosa_t*)arbol_buscar(arbol, auxiliar))->clave==5?"SI":
    "NO");
96
97          auxiliar->clave = 7;
98          printf("Borro nodo hoja (7): %s\n", (arbol_borrar(arbol, auxiliar))==0?"SI":"NO");
99
100         auxiliar->clave = 6;
101         printf("Borro nodo con un hijo (6): %s\n", (arbol_borrar(arbol, auxiliar))==0?"SI":"NO");
102
103         auxiliar->clave = 2;
104         printf("Borro nodo con dos hijos (2): %s\n", (arbol_borrar(arbol, auxiliar))==0?"SI":"NO");
105
106         auxiliar->clave = 4;
107         printf("Borro la raiz (4): %s\n", (arbol_borrar(arbol, auxiliar))==0?"SI":"NO");
108
109         auxiliar->clave = 3;
110         printf("Busco el elemento (3): %s\n", ((cosa_t*)arbol_buscar(arbol, auxiliar))->clave==3?"SI
    ":"NO");
111
112         cosa_t* elementos[10];
113
114         printf("Recorrido inorden (deberian salir en orden 1 3 5): ");
115         int cantidad = arbol_recorrido_inorden(arbol, (void**)elementos, 10);
```

```c
        for(int i=0;i<cantidad;i++)
                printf("%i ", elementos[i]->clave);
        printf("\n");

        printf("\n\nInserto mas valores y pruebo el iterador interno\n\n");
        arbol_insertar(arbol, crear_cosa(15));
        arbol_insertar(arbol, crear_cosa(0));
        arbol_insertar(arbol, crear_cosa(9));
        arbol_insertar(arbol, crear_cosa(7));
        arbol_insertar(arbol, crear_cosa(4));

        printf("Recorrido inorden iterador interno: ");
        abb_con_cada_elemento(arbol, ABB_RECORRER_INORDEN, mostrar_elemento, NULL);
        printf("\n");

        printf("Recorrido preorden iterador interno: ");
        abb_con_cada_elemento(arbol, ABB_RECORRER_PREORDEN, mostrar_elemento, NULL);
        printf("\n");

        printf("Recorrido postorden iterador interno: ");
        abb_con_cada_elemento(arbol, ABB_RECORRER_POSTORDEN, mostrar_elemento, NULL);
        printf("\n");

        printf("\nRecorrido inorden hasta encontrar el 5: ");
        abb_con_cada_elemento(arbol, ABB_RECORRER_INORDEN, mostrar_hasta_5, NULL);
        printf("\n");

        printf("Recorrido preorden hasta encontrar el 5: ");
        abb_con_cada_elemento(arbol, ABB_RECORRER_PREORDEN, mostrar_hasta_5, NULL);
        printf("\n");

        printf("Recorrido postorden hasta encontrar el 5: ");
        abb_con_cada_elemento(arbol, ABB_RECORRER_POSTORDEN, mostrar_hasta_5, NULL);
        printf("\n");

        int acumulador=0;
        printf("\nRecorrido inorden acumulando los valores: ");
        abb_con_cada_elemento(arbol, ABB_RECORRER_INORDEN, mostrar_acumulado, &acumulador);
        printf("\n");

        acumulador=0;
        printf("Recorrido preorden acumulando los valores: ");
        abb_con_cada_elemento(arbol, ABB_RECORRER_PREORDEN, mostrar_acumulado, &acumulador);
        printf("\n");

        acumulador=0;
        printf("Recorrido postorden acumulando los valores: ");
        abb_con_cada_elemento(arbol, ABB_RECORRER_POSTORDEN, mostrar_acumulado, &acumulador);
        printf("\n");

        free(auxiliar);
        arbol_destruir(arbol);

        return 0;
}
```

La salida por pantalla luego de correrlas con valgrind debería ser:

```
==8560== Memcheck, a memory error detector
==8560== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==8560== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==8560== Command: ./abb
==8560==
El nodo raiz deberia ser 4: SI
Busco el elemento 5: SI
Borro nodo hoja (7): SI
Borro nodo con un hijo (6): SI
Borro nodo con dos hijos (2): SI
Borro la raiz (4): SI
Busco el elemento (3): SI
Recorrido inorden (deberian salir en orden 1 3 5): 1 3 5


Inserto mas valores y pruebo el iterador interno

Recorrido inorden iterador interno: 0 1 3 4 5 7 9 15
Recorrido preorden iterador interno: 3 1 0 5 4 15 9 7
```

```
Recorrido postorden iterador interno: 0 1 4 7 9 15 5 3

Recorrido inorden hasta encontrar el 5: 0 1 3 4 5
Recorrido preorden hasta encontrar el 5: 3 1 0 5
Recorrido postorden hasta encontrar el 5: 0 1 4 7 9 15 5

Recorrido inorden acumulando los valores: 0 1 4 8 13 20 29 44
Recorrido preorden acumulando los valores: 3 4 4 9 13 28 37 44
Recorrido postorden acumulando los valores: 0 1 5 12 21 36 41 44
==8560==
==8560== HEAP SUMMARY:
==8560==     in use at exit: 0 bytes in 0 blocks
==8560==   total heap usage: 27 allocs, 27 frees, 1,544 bytes allocated
==8560==
==8560== All heap blocks were freed -- no leaks are possible
==8560==
==8560== For counts of detected and suppressed errors, rerun with: -v
==8560== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

5. Entrega

La entrega deberá contar con todos los archivos necesarios para compilar y ejecutar correctamente el TDA.

Dichos archivos deberán formar parte de un único archivo **.zip** el cual será entregado a través de la plataforma de corrección automática **Kwyjibo**.

El archivo comprimido deberá contar, además del TDA con:

- El archivo con las pruebas agregadas para comprobar el correcto funcionamiento del TDA. En este trabajo es importante realizar pruebas tanto de caja negra como de caja blanca.
- Un **Readme.txt** donde se deberá explicar qué es lo entregado, como compilarlo (línea de compilación), como ejecutarlo (línea de ejecución) y todo lo que crea necesario aclarar. Adicionalmente el archivo debe poseer una sección donde se desarrollen los siguientes conceptos:
  - Explicar qué es un **ABB** y cómo se diferencia de un **Árbol Binario**.
  - Explicar cuál es el objetivo de tener una función de destrucción en el **TDA** y qué implicaría para el usuario no tenerla.
  - ¿Cuál es la complejidad de las diferentes operaciones del ABB? Justifique.
- El enunciado.