

75.41 Algoritmos y Programación II Curso 4

TDA Arbol

Binario de Búsqueda

8 de octubre de 2019

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

```
1 #ifndef __ARBOL_BINARIO_DE_BUSQUEDA_H__
2 #define __ARBOL_BINARIO_DE_BUSQUEDA_H__
3
4 #define ABB_RECORRER_INORDEN 0
5 #define ABB_RECORRER_PREORDEN 1
6 #define ABB_RECORRER_POSTORDEN 2
7
8 #include <stdbool.h>
9 #include <stdlib.h>
10
11 /*
12  * Comparador de elementos. Recibe dos elementos del arbol y devuelve
13  * 0 en caso de ser iguales, 1 si el primer elemento es mayor al
14  * segundo o -1 si el primer elemento es menor al segundo.
15  */
16 typedef int (*abb_comparador)(void*, void*);
17
18 /*
19  * Destructor de elementos. Cada vez que un elemento deja el arbol
20  * (arbol_borrar o arbol_destruir) se invoca al destructor pasandole
21  * el elemento.
22  */
23 typedef void (*abb_liberar_elemento)(void*);
24
25 typedef struct nodo_abb {
26     void* elemento;
27     struct nodo_abb* izquierda;
28     struct nodo_abb* derecha;
29 } nodo_abb_t;
30
31 typedef struct abb{
32     nodo_abb_t* nodo_raiz;
33     abb_comparador comparador;
34     abb_liberar_elemento destructor;
35 } abb_t;
36
37 /*
38  * Crea el arbol y reserva la memoria necesaria de la estructura.
39  * Comparador se utiliza para comparar dos elementos.
40  * Destructor es invocado sobre cada elemento que sale del arbol,
41  * puede ser NULL indicando que no se debe utilizar un destructor.
42  *
43  * Devuelve un puntero al arbol creado o NULL en caso de error.
44  */
45 abb_t* arbol_crear(abb_comparador comparador, abb_liberar_elemento destructor);
```

```
46
47 /*
48  * Inserta un elemento en el arbol.
49  * Devuelve 0 si pudo insertar o -1 si no pudo.
50  * El arbol admite elementos con valores repetidos.
51  */
52 int arbol_insertar(abb_t* arbol, void* elemento);
53
54 /*
55  * Busca en el arbol un elemento igual al provisto (utilizando la
56  * funcion de comparación) y si lo encuentra lo quita del arbol.
57  * Adicionalmente, si encuentra el elemento, invoca el destructor con
58  * dicho elemento.
59  * Devuelve 0 si pudo eliminar el elemento o -1 en caso contrario.
60  */
61 int arbol_borrar(abb_t* arbol, void* elemento);
62
63 /*
64  * Busca en el arbol un elemento igual al provisto (utilizando la
65  * funcion de comparación).
66  *
67  * Devuelve el elemento encontrado o NULL si no lo encuentra.
68  */
69 void* arbol_buscar(abb_t* arbol, void* elemento);
70
71 /*
72  * Devuelve el elemento almacenado como raiz o NULL si el árbol está
73  * vacío.
74  */
75 void* arbol_raiz(abb_t* arbol);
76
77
78 /*
79  * Determina si el árbol está vacío.
80  * Devuelve true si lo está, false en caso contrario.
81  */
82 bool arbol_vacio(abb_t* arbol);
83
84 /*
85  * Llena el array del tamaño dado con los elementos de arbol
86  * en secuencia inorden.
87  * Devuelve la cantidad de elementos del array que pudo llenar (si el
88  * espacio en el array no alcanza para almacenar todos los elementos,
89  * llena hasta donde puede y devuelve la cantidad de elementos que
90  * pudo poner).
91  */
92 int arbol_recorrido_inorden(abb_t* arbol, void** array, int tamano_array);
93
94 /*
95  * Llena el array del tamaño dado con los elementos de arbol
96  * en secuencia preorden.
97  * Devuelve la cantidad de elementos del array que pudo llenar (si el
98  * espacio en el array no alcanza para almacenar todos los elementos,
99  * llena hasta donde puede y devuelve la cantidad de elementos que
100  * pudo poner).
101  */
102 int arbol_recorrido_preorden(abb_t* arbol, void** array, int tamano_array);
103
104 /*
105  * Llena el array del tamaño dado con los elementos de arbol
106  * en secuencia postorden.
107  * Devuelve la cantidad de elementos del array que pudo llenar (si el
108  * espacio en el array no alcanza para almacenar todos los elementos,
109  * llena hasta donde puede y devuelve la cantidad de elementos que
110  * pudo poner).
111  */
112 int arbol_recorrido_postorden(abb_t* arbol, void** array, int tamano_array);
113
114 /*
115  * Destruye el arbol liberando la memoria reservada por el mismo.
116  * Adicionalmente invoca el destructor con cada elemento presente en
117  * el arbol.
118  */
119 void arbol_destruir(abb_t* arbol);
120
121 /*
```

```

122 * Iterador interno. Recorre el arbol e invoca la funcion con cada
123 * elemento del mismo. El puntero 'extra' se pasa como segundo
124 * parámetro a la función. Si la función devuelve true, se finaliza el
125 * recorrido aun si quedan elementos por recorrer. Si devuelve false
126 * se sigue recorriendo mientras queden elementos.
127 * El recorrido se realiza de acuerdo al recorrido solicitado. Los
128 * recorridos válidos son: ABB_RECORRER_INORDEN, ABB_RECORRER_PREORDEN
129 * y ABB_RECORRER_POSTORDEN.
130 */
131 void abb_con_cada_elemento(abb_t* arbol, int recorrido, bool (*funcion)(void*, void*), void*
    extra);
132
133 #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:

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

Ejecución:

```
1 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:

```

1 #include "abb.h"
2 #include <stdio.h>
3
4 typedef struct cosa{
5     int clave;
6     char contenido[10];
7 }cosa;
8
9 cosa* crear_cosa(int clave){
10     cosa* c = (cosa*)malloc(sizeof(cosa));
11     if(c)
12         c->clave = clave;
13     return c;
14 }
15
16 void destruir_cosa(cosa* c){
17     if(c)
18         free(c);
19 }
20
21 int comparar_cosas(void* elemento1, void* elemento2){
22     if(!elemento1 || !elemento2)
23         return 0;
24
25     if(((cosa*)elemento1)->clave > ((cosa*)elemento2)->clave)
26         return 1;
27     if(((cosa*)elemento1)->clave < ((cosa*)elemento2)->clave)
28         return -1;
29     return 0;
30 }
31
32 void destructor_de_cosas(void* elemento){
33     if(!elemento)
34         return;
35     destruir_cosa((cosa*)elemento);
36 }
37

```

```

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

```

```

109         printf("%i ", elementos[i]->clave);
110     printf("\n");
111
112     printf("\n\nInserto mas valores y pruebo el iterador interno\n\n");
113     arbol_insertar(arbol, crear_cosa(15));
114     arbol_insertar(arbol, crear_cosa(0));
115     arbol_insertar(arbol, crear_cosa(9));
116     arbol_insertar(arbol, crear_cosa(7));
117     arbol_insertar(arbol, crear_cosa(4));
118
119     printf("Recorrido inorden iterador interno: ");
120     abb_con_cada_elemento(arbol, ABB_RECORRER_INORDEN, mostrar_elemento, NULL);
121     printf("\n");
122
123     printf("Recorrido preorden iterador interno: ");
124     abb_con_cada_elemento(arbol, ABB_RECORRER_PREORDEN, mostrar_elemento, NULL);
125     printf("\n");
126
127     printf("Recorrido postorden iterador interno: ");
128     abb_con_cada_elemento(arbol, ABB_RECORRER_POSTORDEN, mostrar_elemento, NULL);
129     printf("\n");
130
131     printf("\nRecorrido inorden hasta encontrar el 5: ");
132     abb_con_cada_elemento(arbol, ABB_RECORRER_INORDEN, mostrar_hasta_5, NULL);
133     printf("\n");
134
135     printf("Recorrido preorden hasta encontrar el 5: ");
136     abb_con_cada_elemento(arbol, ABB_RECORRER_PREORDEN, mostrar_hasta_5, NULL);
137     printf("\n");
138
139     printf("Recorrido postorden hasta encontrar el 5: ");
140     abb_con_cada_elemento(arbol, ABB_RECORRER_POSTORDEN, mostrar_hasta_5, NULL);
141     printf("\n");
142
143     int acumulador=0;
144     printf("\nRecorrido inorden acumulando los valores: ");
145     abb_con_cada_elemento(arbol, ABB_RECORRER_INORDEN, mostrar_acumulado, &acumulador);
146     printf("\n");
147
148     acumulador=0;
149     printf("Recorrido preorden acumulando los valores: ");
150     abb_con_cada_elemento(arbol, ABB_RECORRER_PREORDEN, mostrar_acumulado, &acumulador);
151     printf("\n");
152
153     acumulador=0;
154     printf("Recorrido postorden acumulando los valores: ");
155     abb_con_cada_elemento(arbol, ABB_RECORRER_POSTORDEN, mostrar_acumulado, &acumulador);
156     printf("\n");
157
158     free(auxiliar);
159     arbol_destruir(arbol);
160     return 0;
161 }

```

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

```

1 ==8560== Memcheck, a memory error detector
2 ==8560== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
3 ==8560== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
4 ==8560== Command: ./abb
5 ==8560==
6 El nodo raiz deberia ser 4: SI
7 Busco el elemento 5: SI
8 Borro nodo hoja (7): SI
9 Borro nodo con un hijo (6): SI
10 Borro nodo con dos hijos (2): SI
11 Borro la raiz (4): SI
12 Busco el elemento (3): SI
13 Recorrido inorden (deberian salir en orden 1 3 5): 1 3 5
14
15
16 Inserto mas valores y pruebo el iterador interno
17
18 Recorrido inorden iterador interno: 0 1 3 4 5 7 9 15
19 Recorrido preorden iterador interno: 3 1 0 5 4 15 9 7
20 Recorrido postorden iterador interno: 0 1 4 7 9 15 5 3
21

```

```
22 Recorrido inorden hasta encontrar el 5: 0 1 3 4 5
23 Recorrido preorden hasta encontrar el 5: 3 1 0 5
24 Recorrido postorden hasta encontrar el 5: 0 1 4 7 9 15 5
25
26 Recorrido inorden acumulando los valores: 0 1 4 8 13 20 29 44
27 Recorrido preorden acumulando los valores: 3 4 4 9 13 28 37 44
28 Recorrido postorden acumulando los valores: 0 1 5 12 21 36 41 44
29 ==8560==
30 ==8560== HEAP SUMMARY:
31 ==8560==      in use at exit: 0 bytes in 0 blocks
32 ==8560==    total heap usage: 27 allocs, 27 frees, 1,544 bytes allocated
33 ==8560==
34 ==8560== All heap blocks were freed -- no leaks are possible
35 ==8560==
36 ==8560== For counts of detected and suppressed errors, rerun with: -v
37 ==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.
- El enunciado.