75.41 Algortimos y Programación II Curso 4
TDA Arbol

Binario de Búsqueda

14 de mayo 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.

2. abb.h

```c
#ifndef __ARBOL_BINARIO_DE_BUSQUEDA_H__
#define __ARBOL_BINARIO_DE_BUSQUEDA_H__

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

typedef int (*abb_comparador)(void*, void*);

typedef void (*abb_liberar_elemento)(void*);

typedef struct nodo {
        void* elemento;
        struct nodo* izquierda;
        struct nodo* derecha;
} nodo_t;

typedef struct arbol_binario {
        nodo_t* nodo_raiz;
        abb_comparador comparador;
        abb_liberar_elemento destructor;
} abb_t;

/*
 * Crea el arbol y reserva la memoria necesaria de la estructura.
 */
abb_t* crear_arbol(abb_comparador comparador, abb_liberar_elemento destructor);

/*
 * Devuelve 0 si pudo insertar o -1 si no pudo.
 * No puede insertar la misma clave.
 */
int insertar(abb_t* arbol, void* elemento);

/*
 * Elimina el elemento cuya clave coincide con la enviada.
 * Adicionalmente invoca el destructor con el elemento a eliminar.
 * Devuelve 0 si puede eliminar o -1 si no.
 */
int borrar(abb_t* arbol, void* elemento);

/*
 * Busca un elemento en el arbol.
 *
 * El parametro elemento se utiliza para comparar con los otros
 * elementos del arbol. Debe estar inicializado como para poder
 * utilizar la función de comparación.
 *
 * Devuelve el elemento encontrado o NULL si no lo encuentra.
```

```
 */
void* buscar(abb_t* arbol, void* elemento);

/*
 * Determina si el árbol está vacío.
 * Devuelve true si lo está, false en caso contrario.
 */
bool 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.
 */
int recorrer_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.
 */
int recorrer_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.
 */
int recorrer_postorden(abb_t* arbol, void** array, int tamanio_array);

/*
 * Destruye el arbol liberando la memoria reservada por este
 * y todos sus nodos y hojas. Adicionalmente invoca el destructor
 * con cada elemento presente en el arbol.
 * Devuelve 0 si puede destruir el arbol o -1 si no.
 */
int destruir_arbol(abb_t*);

#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;

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

```
}

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

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

        if(((cosa*)elemento1)->clave>((cosa*)elemento2)->clave)
                return 1;
        if(((cosa*)elemento1)->clave<((cosa*)elemento2)->clave)
                return -1;
        return 0;
}

void destructor_de_cosas(void* elemento){
        if(!elemento)
                return;
        destruir_cosa((cosa*)elemento);
}

int main(){
        abb_t* arbol = crear_arbol(comparar_cosas, destructor_de_cosas);

        cosa* c1= crear_cosa(1);
        cosa* c2= crear_cosa(2);
        cosa* c3= crear_cosa(3);
        cosa* c4= crear_cosa(4);
        cosa* c5= crear_cosa(5);
        cosa* c6= crear_cosa(6);
        cosa* c7= crear_cosa(7);
        cosa* auxiliar = crear_cosa(0);

        insertar(arbol, c4);
        insertar(arbol, c2);
        insertar(arbol, c6);
        insertar(arbol, c1);
        insertar(arbol, c3);
        insertar(arbol, c5);
        insertar(arbol, c7);

        printf("El nodo raiz deberia ser 4: %s\n", ((cosa*)arbol->nodo_raiz->elemento)->clave
    ==4?"SI":"NO");

        auxiliar->clave = 5;
        printf("Busco el elemento 5: %s\n", ((cosa*)buscar(arbol, auxiliar))->clave==5?"SI":"
    NO");

        auxiliar->clave = 7;
        printf("Borro nodo hoja (7): %s\n", (borrar(arbol, auxiliar))==0?"SI":"NO");

        auxiliar->clave = 6;
        printf("Borro nodo con un hijo (6): %s\n", (borrar(arbol, auxiliar))==0?"SI":"NO");

        auxiliar->clave = 2;
        printf("Borro nodo con dos hijos (2): %s\n", (borrar(arbol, auxiliar))==0?"SI":"NO");

        auxiliar->clave = 4;
        printf("Borro la raiz (4): %s\n", (borrar(arbol, auxiliar))==0?"SI":"NO");

        auxiliar->clave = 3;
        printf("Busco el elemento (3): %s\n", ((cosa*)buscar(arbol, auxiliar))->clave==3?"SI":
    "NO");

        cosa* elementos[10];

        printf("Recorrido inorden (deberian salir en orden 1 3 5): ");
        int cantidad = recorrer_inorden(arbol, (void**)elementos, 10);
        for(int i=0;i<cantidad;i++)
                printf("%i ", elementos[i]->clave);
        printf("\n");

        free(auxiliar);
```

```
87          destruir_arbol(arbol);
88          return 0;
89  }
```

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

```
1  ==7653== Memcheck, a memory error detector
2  ==7653== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
3  ==7653== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
4  ==7653== Command: ./abb
5  ==7653==
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 ==7653==
15 ==7653== HEAP SUMMARY:
16 ==7653==     in use at exit: 0 bytes in 0 blocks
17 ==7653==   total heap usage: 17 allocs, 17 frees, 1,344 bytes allocated
18 ==7653==
19 ==7653== All heap blocks were freed -- no leaks are possible
20 ==7653==
21 ==7653== For counts of detected and suppressed errors, rerun with: -v
22 ==7653== 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.
- 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.