75.41 95.15 95.12 Algoritmos y Programación II Curso 4
TDA ABB

Árbol Binario de Búsqueda

30 de abril de 2022

1. Enunciado

Se pide implementar una Árbol Binario de Búsqueda (ABB) en el lenguaje de programación C. 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 **ABB** cumpliendo con las buenas prácticas de programación. Adicionalmente se pide la creación de un iterador interno que sea capaz de realizar diferentes recorridos en el árbol y una funcione que guarda la información almacenada en el árbol en un vector.

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.

A **modo de demostración**, se brindará al alumno un archivo simple de "ejemplo". **Este archivo no es mas que un programa mínimo de ejemplo de utilización del TDA a implementar** y es provisto sólo a fines ilustrativos como una ayuda extra para entender el funcionamiento del mismo. No es necesario modificar ni entregar el archivo de minidemo, pero si la implementación es correcta, debería correr con valgrind sin errores de memoria.

Para la resolución de este trabajo se recomienda utilizar una **metodología orientada a pruebas**. A tal fín, se incluye un archivo **pruebas.c** que debe ser completado con las pruebas pertinentes de cada una de las diferentes primitivas del TDA. **El archivo de pruebas forma parte de la entrega y por lo tanto de la nota final.** Aún mas importante, las pruebas van a resultar fundamentales para lograr no sólamente una implementación correcta, si no también una experiencia de desarrollo menos turbulenta.

2. Aclaraciones de la implementación

Esta implementación de **ABB** incluye una función de comparación que permite insertar cualquier tipo de puntero en el mismo. **El ABB no tiene idea de qué es lo que almacena el usuario**, simplemente es un contenedor de datos que sigue reglas definidas. Mediante el comparador el **ABB** es capaz de establecer la relación de orden de los diferentes punteros, sin necesidad de tener información extra. **Recuerde que los comparadores devuelven 0, $>$0 o $<$0** según la relación de los elementos comparados (no 0, 1, -1).

Otra característica distintiva de esta implementación es el uso de **funciones de destrucción**. Recuerde que el **ABB** no es mas que un contenedor de datos y no conoce nada acerca del contenido del mismo. Recuerde también que en **C**, quien reserva la memoria es el responsable de liberarla. Brindándole una función de destrucción al **ABB**, podemos pedirle al mismo que se haga cargo de destruir los elementos que almacena una vez que este es destruido. Tenga en cuenta que si el usuario de la implementación no quiere delegar esta responsabilidad al **ABB** puede pasarle una función de destrucción **NULL** o directamente invocar al destructor simple.

Por último, tenga en cuenta que las pruebas de **Chanutron't** suponen que la implementación acomoda los elementos menores del ABB del lado izquierdo y los mayores del lado derecho, y que al borrar nodos con dos hijos no nulos, se reemplaza dicho nodo con el predecesor inorden.

3. Consejos para la elaboración del trabajo

Intente comprender primero el funcionamiento de los nodos con doble puntero del árbol. Proponga una lista de elementos y dibuje a mano (lápiz y papel) varias operaciones de inserción y eliminación de un árbol. Dibuje cada uno de los nodos con sus punteros y datos. Una vez hecho el dibujo, intente aplicar sobre el mismo cada una de las operaciones propuestas para el TDA. Poder dibujar el problema exitosamente y entender cómo funciona le va a ser fundamental a la hora de la implementación.

Empiece la implementación de cada primitiva escribiendo una prueba. A la hora de escribir cada prueba pregúntese primero cuál es el comportamiento correcto de la primitiva en cuestión. Elabore una prueba en base a ese conocimiento. Luego implemente el código de la primitiva que satisfaga esa prueba. Por último pregúntese qué casos erróneos

conocen que posibles entradas pueden hacer que la primitiva falle. Implemente pruebas para estos casos y si es necesario, modifique la implementación para que el funcionamiento sea el correcto.

Recuerde al escribir pruebas: **no se busca en el código de pruebas la encapsulación ni simplificación de las mismas**, no es incorrecto tener pruebas con código repetitivo. Las pruebas son una **especificación** del comportamiento deseado de las primitivas. Como tal, deben ser fáciles de leer y entender su objetivo.

En general, para todo el código: utilice nombres claros de variables y funciones auxiliares. Una variable con el nombre **cantidad_elementos_recorridos** o incluso **cantidad** es mucho mas claro que una variable con el nombre **n**, **cant**, o **rec**. Intente darle un significado el nombre de cada variable, dentro de lo posible (por supuesto, quedan excluidos casos como i, j, etc para bucles y cosas así).

NO escriba código a lo loco sin compilar cada tanto. Implemente la solución de a poco y compilando a cada paso. Dejar la compilación para el final es uno de los peores errores que puede cometer. Para la compilación del trabajo se provee un **Makefile**. Utilice el comando **make** frecuentemente para compilar y correr su programa.

NO avance en la implementación si le quedan errores sin resolver en alguna prueba. Cada vez que escriba una prueba implemente toda la funcionalidad necesaria para que funcione correctamente. Esto incluye liberar memoria y accesos inválidos a la misma. Sólamente una vez que haya logrado que la prueba pase exitosamente es que puede comenzar a escribir la próxima prueba para continuar el trabajo.

NO está permitido modificar el archivo **.h**. Puede hacer modificaciones al **makefile** si lo desea, pero recuerde que **el trabajo será compilado por el sistema de entregas con el header y makefile original**.

4. Entrega

La entrega deberá contar con todos los archivos que se adjuntan con este enunciado (todo lo necesario para compilar y correr correctamente). Dichos archivos deberán formar parte de un único archivo **.zip** el cual será entregado a través de la plataforma de corrección automática **Chanutron't**.

El archivo comprimido deberá contar, además de los archivos del TDA con informe que elabore los siguientes puntos:

- Explique teóricamente qué es una árbol, árbol binario y árbol binario de búsqueda. Explique cómo funcionan, sus operaciones (un análisis de complejidad de cada una de ellas) y por qué es importante la distinción de cada uno de estos diferentes tipos de árboles. Ayúdese con diagramas para explicar.
- Explique su implementación y decisiones de diseño (por ejemplo, si tal o cuál funciones se plantearon de forma recursiva, iterativa o mixta y por qué, que dificultades encontró al manejar los nodos y punteros, reservar y liberar memoria, etc).