FACULTAD DE INGENIERIA
Universidad de Buenos Aires

TDA ABB.

[7541/9515] Algoritmos y Programación II

Primer cuatrimestre de 2022

Alumno: Martín Abramovich
Padrón: 108762
Email: mabramovich@fi.uba.ar

Contenido

1. Introducción

Se solicita implementar un árbol binario de búsqueda (en adelante TDA ABB). Para ello se cuenta con las firmas de las funciones públicas a implementar. Adicionalmente se pide la creación de un iterador interno que sea capaz de realizar diferentes recorridos en el árbol y una función que guarde la información almacenada en el árbol en un vector.
Se implementa además una metodología orientada a pruebas, en que se solicita la creación de pruebas pertinentes para cada una de las diferentes primitivas del TDA.

2. Conceptos teóricos: árbol, árbol binario y árbol binario de búsqueda

Los **árboles** nacen de querer optimizar la búsqueda lineal de la lista (*complejidad O(n)*). Para ello se buscó una estructura que, similar a la búsqueda binaria, permitiría el acceso a la información en un tiempo de O(log n).
Los árboles poseen una naturaleza altamente recursiva. Existen varios tipos, entre ellos los n-arios o generales, binarios, binarios de búsqueda, AVL, Rojo-Negro, etc.

Un árbol es una colección de nodos. Esta colección puede estar vacía o puede tener un nodo raíz que puede tener 0 o varios sub-árboles no vacíos conectados a la raíz, donde cada uno de ellos contiene a una raíz.

El nodo raíz de cada sub-árbol es denominado nodo hijo del nodo r (raíz), y r es el nodo padre de cada sub-árbol de r.

No se puede saber la complejidad de operar en un árbol n-ario ya que no hay un criterio general para, por ejemplo, insertar. Lo mismo con el resto de las operaciones.

Los **árboles binarios** están íntimamente relacionados a las operaciones de búsqueda, con el objetivo de aproximarse a la búsqueda binaria. La particularidad es que el nodo raíz está solamente conectado a dos sub-árboles, lo cual permite determinar la noción de izquierda y derecha.

Sus operaciones son: crear, insertar, borrar, buscar, vacío, destruir, recorrer.

Hay distintas formas de recorrer los árboles binarios, entre las que se encuentran:

- PREORDEN: Primero se visita el nodo actual, luego el sub-árbol izquierdo y luego el derecho.
- INORDEN: Primero se visita el sub-árbol izquierdo, luego el nodo actual y por último el sub-árbol derecho.
- POSTORDEN: Primero se visita el sub-árbol izquierdo, luego el derecho y después el nodo actual.

A un **árbol binario de búsqueda** (ABB) se le agrega un comparador y cada nodo del árbol posee un valor o una clave única. Esto permite determinar que, dado un nodo raíz, los sub-árboles derechos contienen valores mayores que la raíz y los izquierdos valores menores. Además, los sub-árboles también son abb's.

El orden en el que se insertan los elementos en un ABB tiene gran importancia. Dos ABB con los mismos elementos pero insertados en distinto orden no necesariamente son iguales, debido a esto los arboles pueden degenerar en listas y es por esto que las complejidades algorítmicas de las operaciones de buscar, insertar y borrar en un ABB, en el peor de los casos, es O(N). En el mejor de los casos, O(log n) ya que al tener entre 0 y 2 hijos, al hacer una operación se va a izquierda o derecha.

3. Implementación

Gran parte de mi implementación es recursiva. Un comentario que quiero hacer al respecto es la diferencia de las firmas entre las funciones del .h y las recursivas desarrolladas por mí, recibiendo las primeras un abb_t. Bajo la necesidad de recorrer los nodos del árbol que son de tipo *nodo_abb_t* fue que decidí implementar mis funciones recursivas recibiendo este tipo de dato y no un *abb_t*. Esta fue uno de los puntos que me detuvo un poco hasta que encontré esta manera de encararlo.

3.1. Crear ABB

Tal como se mencionó en el apartado teórico, un abb precisa de un comparador. Por esto, para poder crear un abb verifiqué que este comparador sea válido. De ser así, creo el árbol con un *calloc* para que los campos queden inicializados en 0/NULL.
Luego hago la verificación del *calloc* y, si se creó el árbol correctamente, le asigno al campo *comparador* el comparador recibido.

3.2. ABB insertar

Elegí implementar esta función de manera recursiva ya que me pareció la forma más cómoda de hacerlo.
Para esto, creé una función *abb_insertar_recursivo* que compara el elemento a insertar con el elemento contenido en el nodo. Si es menor va para la izquierda y si es mayor para la derecha, hasta que el nodo sea NULL. Llegado este caso, inserta el elemento y aumenta el tamaño del árbol. Si el abb estaba vacío, inserta el elemento como raíz.

3.3. ABB quitar

Esta primitiva también la implementé de manera recursiva. Para empezar verifiqué que exista el árbol y que éste no esté vacío. De lo contrario, retorna NULL.
La función *abb_quitar_recursivo* va recorriendo el árbol hasta encontrar, con el comparador, el elemento a quitar. Si lo encuentra y el nodo a quitar tiene dos hijos, se reemplaza dicho nodo con el predecesor inorden, lo que implementé en la función *extraer_mas_derecho*. Al quitar disminuyo el tamaño del árbol y realizo el *free* del nodo extraído. Si solo tiene un hijo, el nodo quitado se reemplaza por su hijo. Si no tiene hijos, simplemente se borra el nodo.
De no encontrar el elemento a quitar, la función retorna NULL.

3.4. ABB buscar

También implementado de manera recursiva, se recorre el árbol y se va comparando el elemento del nodo con el elemento a buscar. Según esto se va hacia el sub-árbol izquierdo o derecho. De encontrarlo, devuelve el elemento y caso contrario (se llegó a un nodo nulo) devuelve NULL.

3.5. ABB vacío y ABB tamaño

abb_tamaño devuelve el tamaño de un árbol accediendo al campo *tamanio* del *abb_t.* *abb_vacio* devuelve true si el árbol es nulo o, usando abb_tamanio, si el valor que retorna ésta última es 0. Caso contrario, devuelve false.

3.6. ABB destruir y ABB destruir todo

Para *abb_destruir_todo* utilicé nuevamente recursividad. Voy recorriendo el árbol y destruyendo los elementos de los nodos con el destructor recibido y luego haciendo *free* de los nodos. Finalmente, hago *free* del árbol.
Para *abb_destruir* llamé a *abb_destruir_todo* pasándole NULL como destructor.

3.7. ABB con cada elemento

Para desarrollar esta función, primero verifiqué que tanto la función como el árbol recibido sean válidos. Luego inicialicé en 0 un contador y en *true* la variable *continuar* de tipo bool.
Luego hice un *switch* que según el recorrido recibido por parámetro recorrería el árbol de manera inorden, preorden o postorden hasta terminar el recorrido o hasta que la función recibida devuelva false. La función retorna la cantidad de veces que fue llamada la función.

3.8. ABB recorrer

Para implementar esta función inicialicé la variable *elementos_almacenados.* Luego, según el recorrido, recorro el árbol y almaceno en el vector los elementos hasta que el vector se llene o se terminen de almacenar los elementos, lo que suceda primero. La función devuelve los elementos que se almacenaron.

4. Diagramas