TDA N°2 — Árbol Binario de Búsqueda (ABB)

[7541/9515] Algoritmos y Programación II

Primer cuatrimestre de 2022

Alumno:	Salluzzi, Luca
Número de padrón:	108088
Email:	lsalluzzi@fi.uba.ar

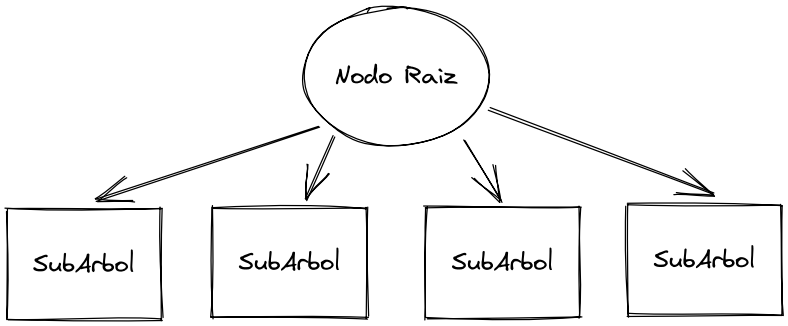
1. Introducción

El ejemplo se corre y se compila usando el comando `make valgrind-ejemplo` y las pruebas mediante el comando `make valgrind-pruebas`
En este trabajo se buscaba afianzar los conceptos teórico-prácticos del TDA Árbol, particularmente del Árbol Binario de Búsqueda. Para esto se nos pidió implementar uno con nodos simplemente enlazados, otorgándonos un .h y un .c con todas las primitivas necesarias y dejando a nuestro criterio la creación de funciones privadas.

2. Teoría

2.1 Árboles

Un árbol puede ser definido de varias formas, la forma más sencilla es de manera recursiva, como un conjunto de nodos enlazados. Lo conforma entonces un nodo particular, el nodo raíz (el cual se distingue del resto por no tener padre) y una cantidad n de nodos hijos. Asimismo, estos nodos hijos serán, a su vez, nodos raíces de sus propios sub-árboles. El nodo raíz de cada sub-árbol es entonces hijo del nodo raíz principal del árbol.



2.11 Árbol Binario

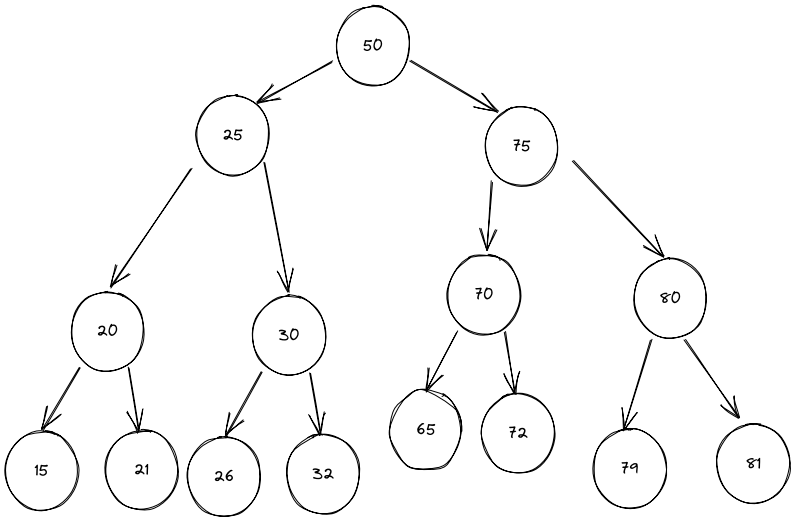
Un árbol binario es un tipo de dato abstracto árbol que puede tener de cero a dos hijos por nodo, denominados izquierdo y derecho para una mayor comprensión por parte del usuario o desarrollador.

2.12 Árbol binario de búsqueda

El árbol binario no tiene mucha utilidad per se, ya que no se establece ninguna regla para la inserción de elementos. Es por esto que en el árbol binario de búsqueda se dispone de un contrato de comparación en el que los elemento mayores al nodo van en su nodo derecho y los menores en el nodo izquierdo. Teniendo siempre un solo valor por nodo.
Entonces, en un árbol binario:

- Si existe nodo izquierdo, va a ser siempre menor en valor a su nodo padre.
- Si existe nodo derecho, va a ser siempre mayor en valor a su nodo padre.

- Los sub-árboles serán también árboles binarios de búsqueda.



2.2 Primitivas y Complejidades Algorítmicas

2.21 Árbol n-ario

Primitivas

Para todo tipo de árbol, las primitivas son:

- crear
- destruir
- vacío
- insertar
- eliminar
- buscar
- recorrer
  Crear, justamente, crea el árbol, lo genera y lo inicializa, según el criterio acordado o la implementación que se desee. En la mayoría de los casos esto significa crear la estructura con nodo raíz con elemento vacío e hijos también vacíos.
  Destruir, borra el árbol, dependiendo del tipo de lenguaje que estemos usando para la implementación, esto puede significar liberar la memoria asignada. Para lo cual es necesario hacerlo en un orden preciso, evitando dejar nodos huérfanos.

Vacío corrobora que el árbol no tenga elementos, esto también se puede pensar como que el tamaño actual del árbol sea cero, pero como siempre, depende de la implementación (ya que puede no tener un contador de tamaño).
Insertar agrega elementos al árbol, dependiendo del contrato/criterio, se agregarán de diferentes formas, cambiando según el tipo de árbol. También, según el caso, puede aumentar el contador de tamaño. Si estamos en C, por ejemplo, se deberá reservar memoria para el nodo del nuevo elemento que estamos agregando.
Eliminar es opuesta a agregar, ya que borra elementos del árbol y disminuye, de ser necesario, el tamaño del mismo. Siguiendo el ejemplo anterior, en programas donde la memoria se maneje manualmente, si esta se encuentra reservada, se deberá liberar la cantidad equivalente al nodo del elemento que estamos eliminando.
Buscar recorre el árbol comparando entre elementos hasta encontrar (o no) el buscado.
Recorrer, finalmente, visita los diferentes nodos del árbol en el orden que se le especifique. Esto puede ser primero el nodo actual y después sus hijos, o primero su primer hijo, después el nodo actual y luego otro hijo, las combinaciones son infinitas si hablamos de árboles n-arios.
Es importante aclarar que muchas de estas primitivas necesitan de un contrato o criterio para su correcto funcionamiento, ya que no se puede insertar, quitar o buscar sin saber como está ordenado y organizado internamente el árbol.

Complejidades algorítmicas

De igual manera, es imposible calcular las complejidades algorítmicas de un árbol n-ario sin ningún tipo de criterio acordado.

2.22 Árbol binario

Un árbol binario comparte las mismas primitivas que los árboles en general, contando con la peculiaridad de que solo se pueden tener dos hijos por nodo, es decir, un máximo de 2 y un mínimo de cero. Si bien esto caracteriza e

individualiza más al tipo de árbol, no es suficiente para establecer un criterio de inserción, eliminación o búsqueda. Más si lo es para poder definir los diferentes tipos de recorridos.
Recorrer es pasar por todos los nodos del árbol, y si definimos al nodo actual como N, al su hijo derecho como D y a su hijo izquierdo como N, existen 6 formas de recorrerlo: NID, IND, IDN, NDI, DNI y DIN. Siendo los 3 estándares NID (preorder), IND (inorden) y IDN (postorden). Siendo la primera una forma de copiar fielmente el árbol, la segunda una forma útil de recorrerlo de menor a mayor y la tercera una forma correcta de eliminar el árbol nodo a nodo.

2.23 Árbol binario de búsqueda

En este tipo de árbol encontramos ya un criterio establecido para el ordenamiento de elementos dentro de la estructura, por lo que, finalmente, las primitivas cobran total sentido.
Tal y como se aclara en la sección 2.12 de este informe,

- Si existe nodo izquierdo, va a ser siempre menor en valor a su nodo padre.
- Si existe nodo derecho, va a ser siempre mayor en valor a su nodo padre.
- Los sub-árboles serán también árboles binarios de búsqueda.
  Esto nos permite no solo entender de forma más clara como se inserta, se quita y se busca (se compara nodo a nodo, desplazándose a la rama derecha o izquierda según el resultado de la comparación), sino también poder establecer las diferentes complejidades algorítmicas de las primitivas.

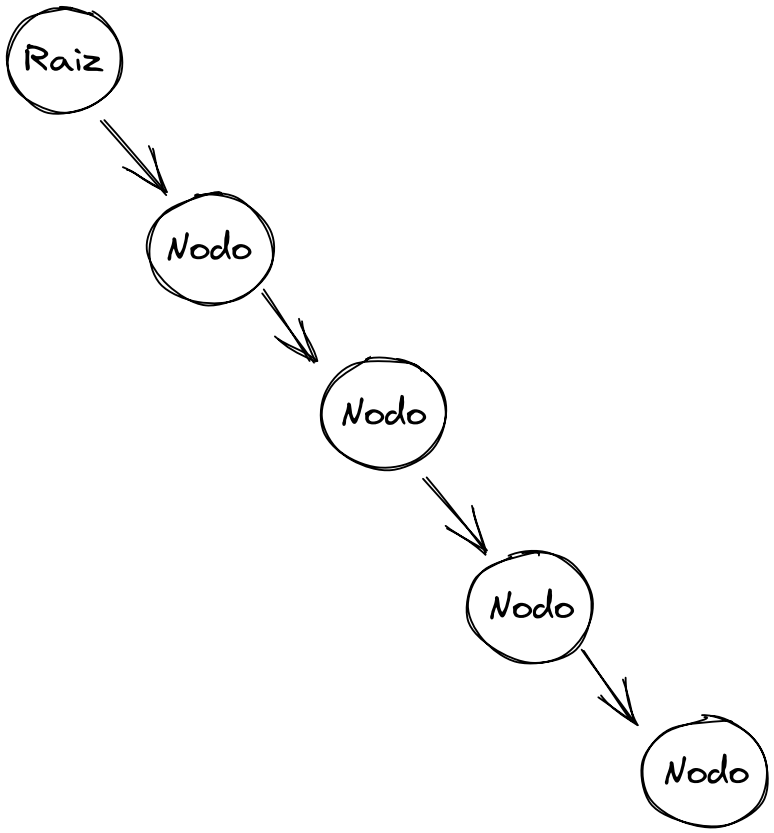
Complejidades algorítmicas

Crear árbol tiene complejidad $O(1)$, ya que siempre, dentro del mismo árbol, crearlo nos va a costar la misma cantidad de memoria. Si queremos crear el árbol con más elementos, deberíamos crear y luego insertar.
Destruir el árbol depende de la cantidad de elementos almacenados, así que su complejidad es $O(n)$. A más elementos insertados, más memoria para borrar y, por ende, más elementos por recorrer.

Si queremos corroborar si el árbol está o no vacío, únicamente deberíamos verificar si la variable contadora es igual o distinta de cero o, de no existir esta, simplemente se debería ver el elemento del nodo raíz. De no existir este, se puede afirmar que el árbol está vacío. Visto de otra forma: si el nodo raíz está vacío, el árbol también lo está. Esto nos lleva a una complejidad algorítmica igual a $O(1)$

Insertar en un árbol binario de búsqueda, la complejidad de la inserción es $O(n)$, ya que estamos considerando el peor caso posible y este es cuando la estructura se degenera a lista y su funcionalidad principal (la de dividir la búsqueda para obtener una mayor eficiencia), se encuentra estropeada. De ser un árbol perfectamente balanceado de manera continua (un AVL, por ejemplo), la complejidad sería constantemente $O(log(n))$

Similarmente, eliminar tiene una complejidad de $O(n)$, por las mismas razones que en la primita anterior, de estar este árbol degenerado y de querer nosotros obtener el último elemento de esa rama, deberíamos antes recorrer todos los elementos anteriores.



Lo mismo ocurre con buscar, teniendo complejidad $O(n)$ en su peor caso dentro de esta implementación.

Finalmente, la complejidad de recorrer siempre va a depender de la cantidad de elementos dentro del árbol, es decir, $O(n)$.

3. Detalles de implementación

En mi implementación, decidí realizar todo de forma recursiva, ya que me pareció lo más natural a la hora de trabajar un Árbol Binario de Búsqueda. Por otra parte, opté por llevar la mayoría de las primitivas que recibían al árbol como parámetro "principal" a primitivas que trabajaran con nodos. De esta manera me fue más fácil operar, puesto que no se genera ninguna distinción entre el nodo raíz y el resto. Entonces, la mayoría de las primitivas de ABB se pueden abstraer al siguiente paso a paso en pseudocódigo

```
Si hay alguna condicion que no va a permitir que la primitiva se
ejecute correctante
evacuar la función devolviendo error
si no-> ejecutar la primitiva con nodos equivalente pasandole el nodo
raiz
```

Por ejemplo, la primitiva de búsqueda

```c
void *abb_buscar(abb_t *arbol, void *elemento)
{
if (arbol == NULL || arbol->tamanio == 0 || arbol->comparador ==
NULL)
return NULL;
return nodo_buscar(arbol->nodo_raiz, elemento, arbol->comparador);
}
```

Luego, en cuanto al cómo se ejecutan las diferentes funciones, intenté que el paso a paso de las mismas se condijera con el pensamiento del programador a la hora de desarrollarla.
De esta forma tenemos, en el caso de `abb_tamanio`

```
Si el arbol es nulo o el tamaño es cero->el arbol está vacio
de lo contrario, está lleno
```

```
bool abb_vacio(abb_t *arbol)
{
if (arbol == NULL || arbol->tamanio == 0)
return true;

return false;
}
```

4. Detalles de Funciones en particular

1. `nodo_quitar()`
   Esta función se encarga de eliminar el elemento pedido, para esto recorre el árbol en forma N I D, es decir, preorden buscando el elemento a eliminar. Una vez encontrado y si tiene hijo izquierdo, busca su predecesor inorder o, visto de otro forma su número menor más cercano. Para esto le pasa el nodo izquierdo a la función `obtener_elemento_mayor()`. Esta se encarga de devolver el elemento más grande de esa rama (la cual es la rama de los números menores al nodo a eliminar). De esta manera, se reemplaza el nodo eliminado con el predecesor y el árbol queda ordenado adecuadamente. Por último, se resta un valor al tamaño del árbol y se libera el nodo donde se alojaba el elemento predecesor, ya que este pasó ahora al lugar del nodo eliminado y nos quedaría un nodo hoja vacío.
   Caso contrario, se guarda el nodo en una variable auxiliar y reemplaza el nodo actual por el nodo derecho. Si este fuese nulo, simplemente quedaría el nodo actual como nulo. Luego, libera el nodo actual. Siempre devuelve el nodo actual al finalizar la función.
   Es gracias a esto último y a las propiedades de la recursividad que esta función mantiene ordenado el árbol. Cuando se encuentra, se elimina el elemento y se empiezan a desapilar en el stack los llamados recursivos,

el retorno de la funcion siempre queda enlazado con el nodo izquierdo o derecho, respectivamente, del nodo anterior, esto se puede visualizar mejor en el extracto de código de abajo.

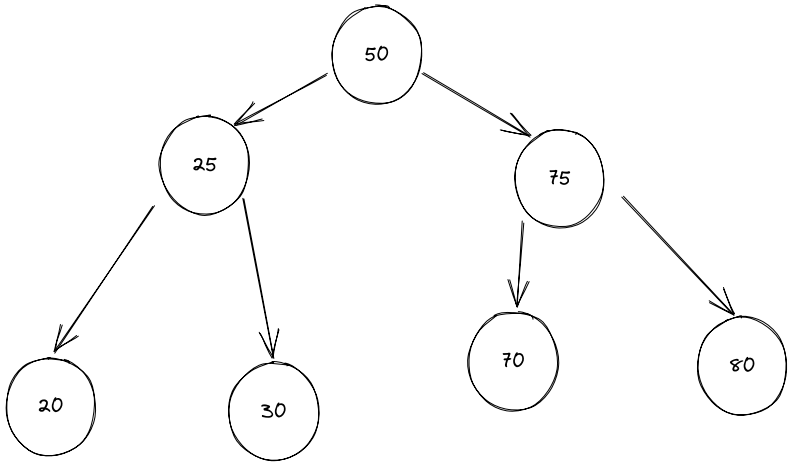
```
if (comparacion < 0)
        nodo->izquierda = nodo_quitar(nodo->izquierda, elemento,
comparador, elemento_quitado, tamanio);

else
        nodo->derecha = nodo_quitar(nodo->derecha, elemento,
comparador, elemento_quitado, tamanio);
```

2. `nodo_destruir_todo()`
   Esta funcion recorre de forma postorder el árbol, aplicando la funcion destructora otorgada por el usuario en cada uno de los elementos y luego liberando el nodo. Se recorre de ese modo en particular (I D N) para nunca dejar nodos huérfanos (aunque, claro está, también se podría recorrer en forma D I N). Si la funcion destructora es nula, simplemente libera los nodos.

5. Diagramas

1. Vector de elementos con recorrido inorden
   Teniendo el siguiente árbol binario de busqueda



   Si lo recorremos con la funcion `abb_recorrer` de manera inorder, la cual

almacena los elementos recorridos en un vector, obtendremos lo siguiente.

2. Si lo recorremos de forma postorder, en cambio, obtendriamos el vector

3. Diagrama de memoria de un arbol binario, con un puntero a un arbol que vive en el heap y que almacena ints que pertenecen al stack