TDA 2 — ABB

[7541/9515] Algoritmos y Programación II
Segundo cuatrimestre de 2021

Alumno:	Castillo, Carlos E.
Número de padrón:	108535
Email:	ccastillo@fi.uba.ar

Índice

1. Introducción

Muchos lenguajes de programación permiten al usuario extender los tipos de datos nativos del lenguaje con para facilitar la implementación de características y funcionalidades más complejas. De esta forma, el programador tiene la capacidad de combinar diferentes primitivas del lenguaje y así generar nuevos tipos compuestos para crear interfaces abstractas que faciliten el manejo y organización de la información mediante diferentes estructuras de datos.

El objetivo de este trabajo práctico es implementar una de las estructuras de datos más populares, el árbol binario de búsqueda.

2. Teoría

2.1. TDA Árbol

Un árbol es una colección de datos en la que la información mantiene una estructura jerárquica. Esta estructura puede ser definida de manera recursiva utilizando nodos enlazados, en la que cada nodo es una estructura que contiene el valor almacenado y referencias a los nodos siguientes, conocidos como **hijos**. Cada árbol tienen un nodo en particular que es inicio o el punto de entrada al árbol, es el único nodo sin padres, y se le conoce como **raiz**. Por otra parte, a aquellos nodos que no tienen ningún hijo, se les conoce como **hojas**.

Existen muchas variantes de esta estructura de datos, cada una con diferentes propiedades y utilidades, por ejemplo:

- Árbol Binario
- Árbol Binario de búsqueda
- Árbol AVL
- Árbol B
- Árbol Rojo Negro

Sin embargo la más popular entre estas implementaciones es el **árbol binario**, que consiste en un árbol cuyos nodos pueden tener un máximo de dos hijos cada uno (esto no impide que un nodo pueda estar vació o tener solo un solo hijo). Para facilitar la diferenciación entre los hijos de un nodo, se les suele llamar ”nodo izquierdo” y ”nodo derecho” respectivamente, en función de la ubicación relativa de cada uno de los nodos con su nodo padre.

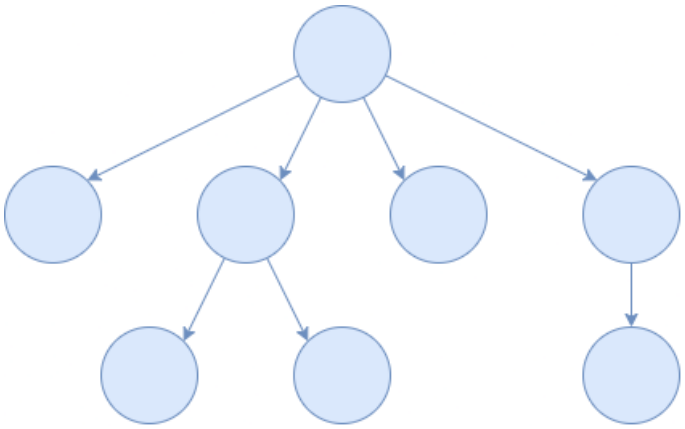

Figura 1: Ejemplo de árbol n-ario.

2.1.1. Árbol Binario de Búsqueda (ABB)

Este tipo de árbol es un caso particular de un árbol binario. Como su nombre bien lo indica, es utilizado principalmente para operaciones de búsqueda entre los datos almacenados. Consiste en un tipo de árbol binario en el que se tiene un criterio de comparación definido para ser utilizado al momento de insertar nuevos nodos al árbol. Esto permite realizar búsquedas de manera más sencilla, ya que al contar con este patrón de ordenamiento, se pueden realizar operaciones de búsqueda binaria, recortando el tiempo de búsqueda en comparación con las listas enlazadas u otras estructuras de datos lineales.

Generalmente se plantea un método de comparación que permite distinguir entre valores considerados menores o mayores al nodo actual, así por ejemplo, si se le asigna la posición izquierda a los elementos cuyo valor es menor al valor del elemento del nodo actual, entonces se sabe que todos los nodos a la izquierda de cualquier nodo van a tener un valor menor al de sus respectivos padres, y todos los nodos la derecha, tendran un valor mayor al de sus padres.

Esto facilita la navegación o **recorrido** del árbol, ya que al saber si el elemento buscado es menor o mayor que el nodo actual, se puede determinar en qué dirección seguir el recorrido y descartar todas aquellas ramas que no satisfagan la comparación.

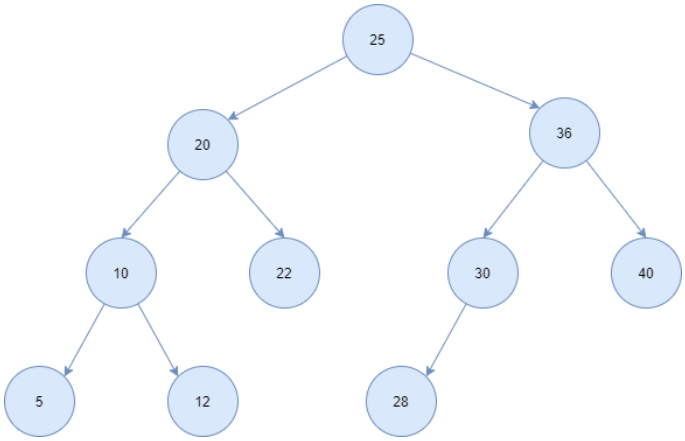

Figura 2: Ejemplo de árbol binario.

3. Detalles de implementación

La implementación de esta estructura de datos fue escrita en el lenguaje de programación C, siguiendo la especificación del lenguaje detallada en el estándar C99. Los archivos principales se encuentran dentro del directorio **src** ubicada en la raíz del repositorio.

Además se incluye un archivo **pruebas.c** en el que se encuentran diferentes pruebas automatizadas para detectar errores en la ejecución del programa o en la asignación, manejo y liberación de memoria dinámica. Para esto se utiliza **gcc** como compilador y **valgrind** como herramienta de análisis de memoria.

La estructura de árbol binario de búsqueda propuesta para esta implementación es la siguiente:

```
typedef struct nodo_abb {
  void* elemento;
  struct nodo_abb* izquierda;
  struct nodo_abb* derecha;
} nodo_abb_t;

typedef struct abb{
  nodo_abb_t* nodo_raiz;
```

```
    abb_comparador comparador;
    size_t tamanio;
} abb_t;
```

Esta estructura no es meramente recursiva, ya que además de contener referencias a otros nodos (en este caso únicamente a la raíz del árbol), almacena el tamaño actual del árbol (la cantidad de nodos que contiene) y una función comparador que establece el criterio de comparación utilizado para el árbol.

Por otra parte, la estructura de cada uno de los nodos si es recursiva. Cada nodo puede ser considerado como un sub-árbol, y almacena el elemento a guardar así como también referencias a sus nodos hijos derecho e izquierdo.

3.1. Inserción de elementos

La función abb_insertar permite agregar nuevos elementos a un ABB. En el caso de que el ABB esté vacio, cuando se inserte el primer elemento, este pasará a ubicarse en el nodo raíz. En este caso la posición en la que se va a insertar el nodo que contiene al nuevo nodo se define a partir del criterio de comparación propio de cada árbol.

La operación de inserción consiste en una serie de 3 pasos:

1. Comparar el valor del elemento a insertar con el valor del nodo raíz. En base al resultado de esta comparación se sabrá si navegar hacia la rama izquierda del árbol (en el caso de que el valor comparado sea menor al de la raíz) o a la derecha (en caso de que el valor comparado se mayor).

2. Se repite el paso anterior con cada uno de los nodos que se vayan recorriendo hasta llegar al final de la rama.

3. Al llegar al nodo final se crea un nuevo nodo que contiene el elemento a insertar y se posiciona en la ubicación correspondiente (izquierda o derecha) de acuerdo con el resultado de la comparación anterior.

En este caso la implementación de la operación de inserción es recursiva. Para esto la función abb_insertar antes mencionada, utiliza una función auxiliar llamada abb_insertar_recursivo_aux. En la primera función solamente se hace la validación de los argumentos y se verifica que el nuevo elemento haya sido insertado con éxito. En caso afirmativo, se incrementa la cantidad de elementos del árbol y finalmente se retorna el nuevo árbol.

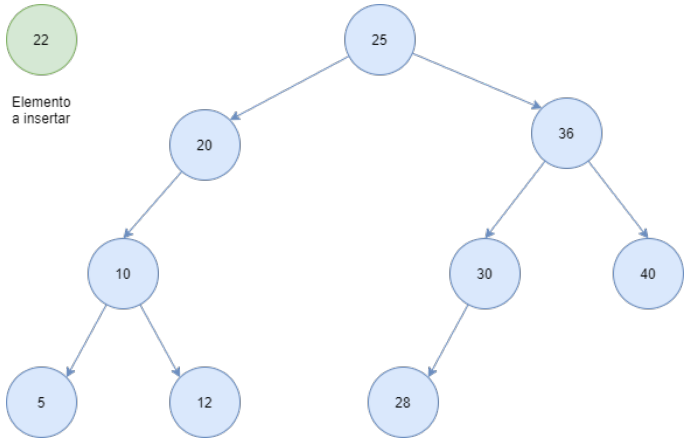

Figura 3: Inserción del elemento de valor 22.

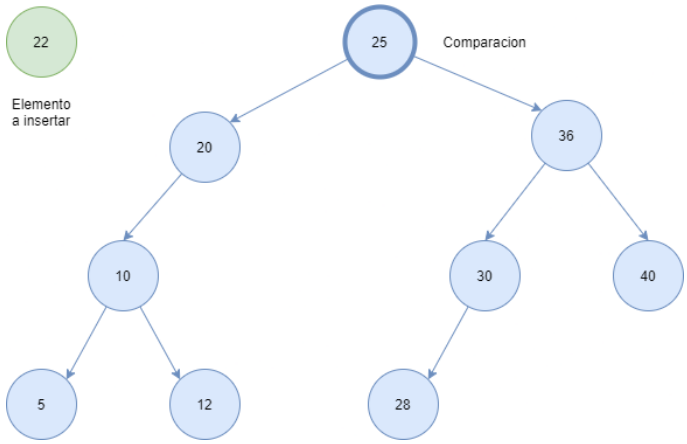

Figura 4: Comparo el valor del elemento con el valor de la raíz.

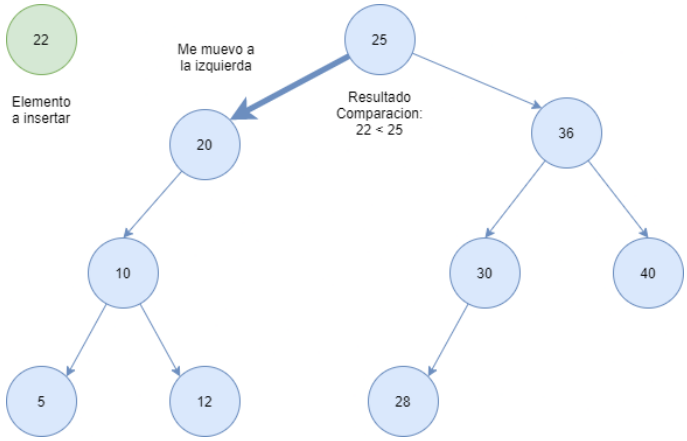

Figura 5: Avanzo hacia el sub-árbol izquierdo ya que el valor del elemento es menor.

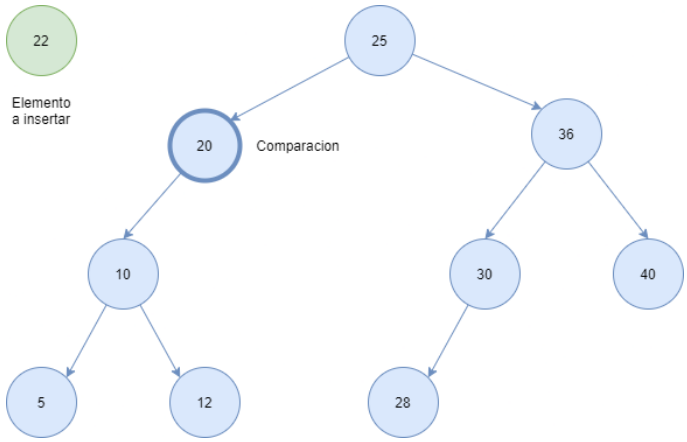

Figura 6: Vuelvo a comparar los valores de los elementos.

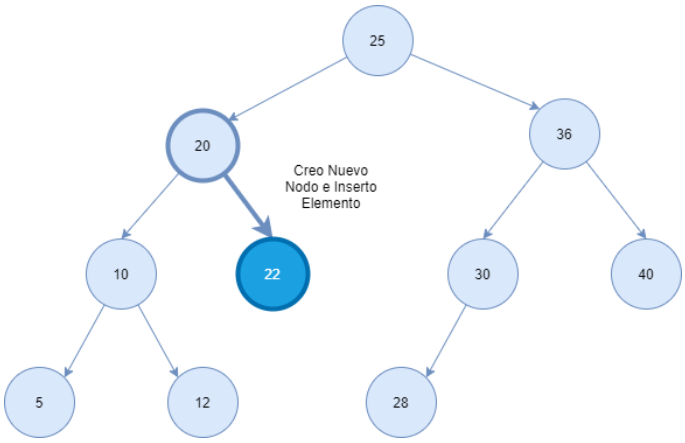

Figura 7: Creo un nuevo hijo izquierdo e inserto el nuevo valor.

3.2. Eliminación de elementos

Para quitar elementos del ABB primero se tiene que realizar un recorrido del mismo para buscar el elemento que se desea eliminar. En el caso en que el elemento que se desea eliminar no esté en el árbol, simplemente se termina la búsqueda. En caso contrario, la operación de eliminación puede ser dividida en tres casos particulares:

- **Nodo hoja:** En el caso en el que el elemento que se quiere eliminar esté ubicado en un nodo hoja, el procedimiento para remover este elemento simplemente consiste en extraer el elemento que contiene ese nodo (para no perder la referencia al elemento) y después la liberación del nodo de la memoria.
- **Nodo con un hijo:** En este caso se tiene que identificar si nodo actual tiene hijo izquierdo o derecho. Luego se enlaza el hijo identificado con el padre del nodo a eliminar, de tal forma que el árbol conserve sus propiedades de ABB. Finalmente se extra el valor almacenado en el nodo a eliminar y se procede a la liberacion del nodo de la memoria.
- **Nodo con dos hijos:** En este último caso se tiene que encontrar el antecesor o sucesor inmediato al nodo eliminar, es decir, para los nodos cuyo valor es menor al valor del nodo a eliminar, extraer el nodo más a la derecha de estos mismos, es decir, el nodo con mayor valor, en el caso de el sucesor inmediato es el caso contrario, se extraer el nodo con menor valor entre todos los nodos que están a la derecha del nodo a eliminar, es decir, entre los nodos cuyo valor es mayor al nodo a eliminar. Una vez encontrado el nodo correcto, se reemplaza por el nodo a eliminar. Esta forma de extracción asegura que el nodo que pasa a tomar la posición del nodo a eliminar permita que se mantenga el orden del ABB. Finalmente se guarda una referencia al valor del nodo a eliminar y se libera de la memoria.

Al finalizar la eliminación del nodo se retorna la referencia al elemento que estaba almacenado en este.

Para reutilizar un poco mejor el código, en esta implementación tanto la eliminación de un nodo hoja como la eliminación de un nodo con un solo hijo se hacen de la misma forma. Estos dos casos pueden ser agrupados en uno solo para nodos que tienen como máximo un nodo. Al encontrar el nodo a eliminar, se sabe que este puede tener un hijo, entonces se busca si este nodo tiene hijo derecho. En el caso de que exista hijo derecho (que este no sea NULL), se guarda una referencia al hijo derecho que luego va a ser enlazada con el padre del nodo a eliminar, en en el caso de que no tenga hijo derecho, entonces se guarda la referencia al hijo izquierda y se enlaza dicho nodo. Si el nodo a eliminar no tenía ningún hijo, entonces igualmente se enlaza el hijo izquierdo, que como no existe, vale NULL, es decir que al eliminar el nodo deseado, el padre de este va a quedar con una referencia a NULL, que es equivalente a directamente eliminar el nodo hoja deseado.

Cabe destacar que al igual que la operación de insertar, la función encargada de la eliminación de elementos también utiliza una función auxiliar recursiva, en este caso esa función es abb_quitar_recursivo_aux. A su vez esta función utiliza un segundo auxiliar abb_extraer_maximo, utilizado para extraer el sucesor inmediato en el caso de eliminación de nodos con dos hijos.

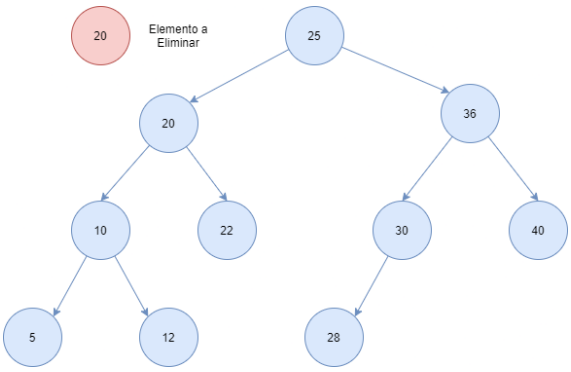

Figura 8: Inicio a buscar el elemento a eliminar desde la raíz.

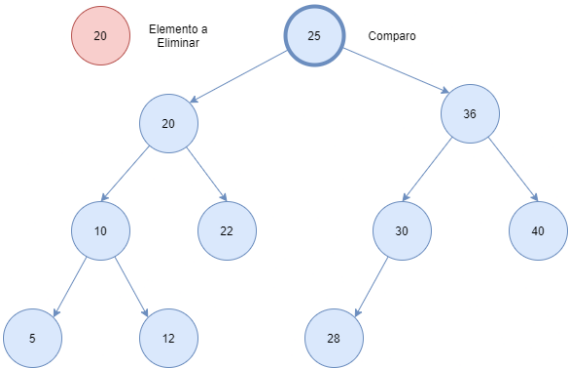

Figura 9: Comparo el valor del elemento con el valor de la raíz.

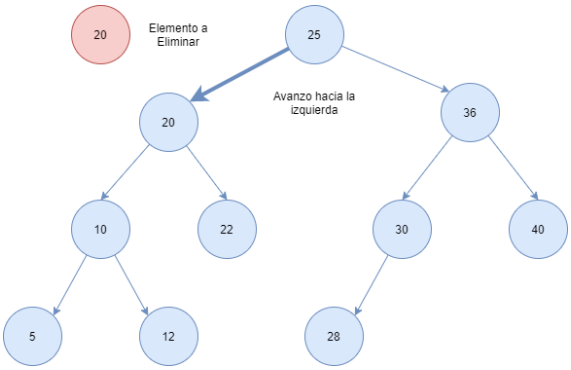

Figura 10: Avanzo hacia el sub-árbol izquierdo ya que el valor del elemento es menor.

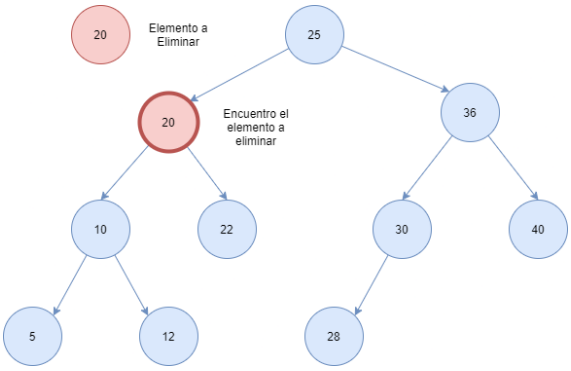

Figura 11: Encuentro el elemento a eliminar.

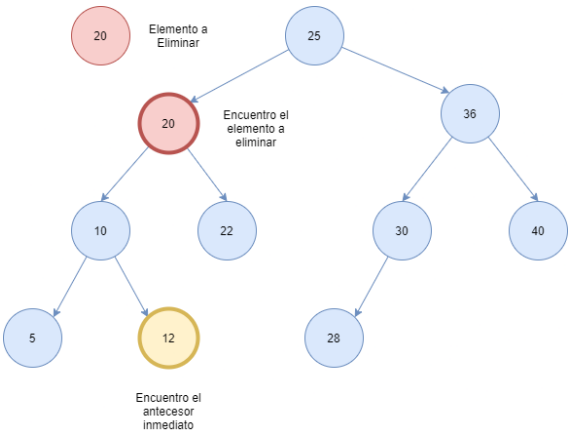

Figura 12: Encuentro el sucesor inmediato al elemento a eliminar.

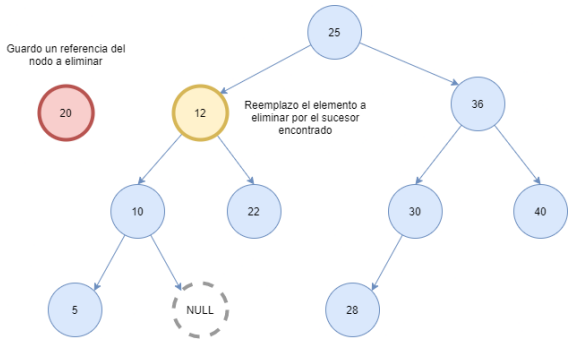

Figura 13: Reemplazo el nodo a eliminar por el sucesor inmediato.

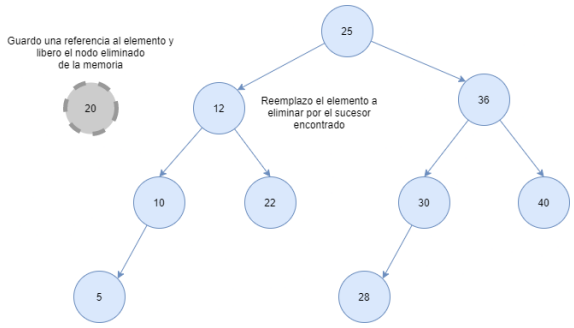

Figura 14: Guardo una referencia al elemento dentro del nodo a eliminar y libero el nodo.

3.3. Búsqueda de un elemento

La operación de búsqueda consiste en una función llamada abb_buscar que a su vez utiliza una función auxiliar recursiva llamada abb_buscar_recursivo_aux. Esta función toma el elemento raíz del árbol y a partir de ahí inicia a comparar los valores de cada nodo con el valor buscado, utilizando la misma función de comparación con la que se insertaron los elementos. En caso de que el valor buscado sea menor al valor del nodo actual, se continúa el recorrido en la rama izquierda, en caso en que el valor buscado sea mayor al valor del nodo actual, la búsqueda se continúa en la derecha, y si los valores son iguales, significa que se encontró el elemento buscado. En este último caso se retorna el valor almacenado.

3.4. Iterador

Existen dos funciones utilizadas para recorrer el ABB. La primera función, llamada abb_recorrer, solamente se encarga de recorrer todos los elementos del árbol en el orden especificado. Para esto se utilizan diferentes funciones auxiliares recursivas que van avanzando a través de los nodos del árbol en según el tipo de recorrido que corresponda. A medida se van recorriendo cada uno de los elementos, estos van siendo agregados a un vector externo en el orden esperado de cada uno de los recorridos.

La segunda de estas funciones, abb_con_cada_elemento es utilizada para aplicar una función sobre cada uno de los elementos del árbol, hasta que se llegue al final del mismo o hasta que la función falle o llegue a un elemento específico, en cuyo caso esta retorna `false`.

Ambas funciones retornan solamente la cantidad de elementos recorridos.

3.5. Getters

Finalmente se proveen una seria de getters para obtener información sobre el estado actual del ABB, como por ejemplo, la función abb_vacio, que retorna `true` o `false` si el ABB está vacío o no, o la función abb_tamanio que retorna la cantidad actual de elementos en el ABB.

3.6. Destrucción del ABB

Cuando se deja de utilizar el ABB, este se libera de la memoria. Para esto existen dos otras dos funciones encargadas de la destrucción del árbol. La primera de estas funciones, llamada abb_destruir_todo, toma un destructor (una función encargada de liberar de la memoria cada uno de los elementos dentro del árbol) y recorre recursivamente el árbol con un recorrido postorden, para ir liberando de la memoria cada nodo hasta luego de haber destruido sus hijos.