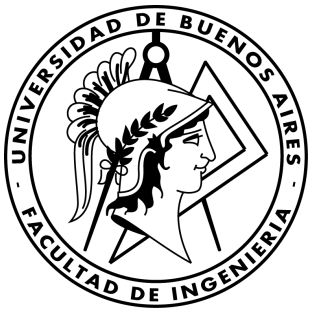


TDA ABB

Algoritmos y Programación II
Segundo cuatrimestre de 2022

Alumno:	Fernández, Facundo Nahuel
Número de padrón:	109097
Email:	ffernandez@fi.uba.ar

Índice

1. Introducción

El objetivo de este trabajo es implementar una abb, creando un iterador interno para dicho abb (capaz de realizar diferentes recorridos en el árbol) e implementando una función que guarde la información almacenada del árbol en un vector. Para compilar el programa (el archivo pruebas.c), se debe acceder desde la terminal a la carpeta en donde se encuentra dicho archivo y escribir el comando ´make´.

En este informe se desarrolla los conceptos de árbol, árbol binario y árbol binario de búsqueda, se explica cómo funcionan las operaciones de inserción, eliminación y búsqueda en cada caso, además de una explicación de cómo funcionan las partes más importantes del código.

2. Teoría

2.1. Árbol

Un árbol es una colección de nodos. Los nodos son los elementos o vértices del árbol que, a su vez, pueden estar conectados a múlriples nodos. Esta colección puede estar vacía. Un árbol consiste en un nodo principal r que se distingue del resto, llamado raíz, y cero o muchos sub-árboles no vacíos $T_1, T_2, ..., T_k$, cada uno de ellos posee su raíz conectada mediante un vértice al nodo raíz r.

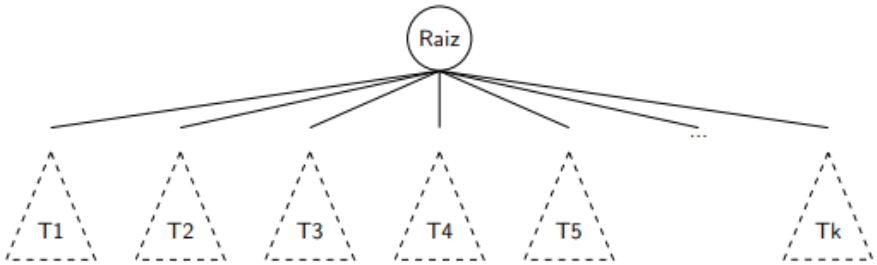


Figura 1: Árbol.

Si estamos parados sobre un nodo, se llama nodo padre al nodo inmediatamente superior y nodos hijos a los nodos conectados a un nivel inferior. Una raíz está conectada a sus subárboles mediante el nodo hijo, y un nodo hijo está conectada a la raíz mediante el nodo padre. Se llama nodo hoja a un nodo sin hijos.

Se crea un árbol con el objetivo de repersentar una jerarquía en la estructua de los datos, así como también de optimizar la búsqueda lineal de una lista (se busca dentro de cada subárbol en vez de linealmente).

Existen diferentes tipos de árboles, entre ellos: n-arios, binarios, binarios de búsqueda (abb), AVL, Rojo-Negro, etc.

Es díficil establecer un método específico de insertar o eliminar elementos del árbol, pues depende de la persona que cree el árbol cómo se establece quiénes son los nodos hijos de la raiz y lo mismo sucederá con los hijos de los hijos. Por el mismo motivo, será díficil buscar elementos dentro del árbol con un método üniversal".

2.2. Árbol Binario

Como se dijo anteriormente, los árboles tienen el objetivo de optimizar la búsqueda dentro de una colección de datos. Los árboles binarios están íntimamente relacionados con las operaciones de búsqueda, con el objetivo de aproximarse a la búsqueda binaria. En un árbol de tipo binario, el nodo raíz está conectado solamente a dos subárboles, y cada nodo tiene un máximo de dos hijos (cada subárbol es un árbol binario).

Cada nodo tiene un puntero a cada uno de sus hijos (izquierdo y derecho). Si no tiene un hijo en alguna de las dos direcciones (o ambas), apunta a NULL.

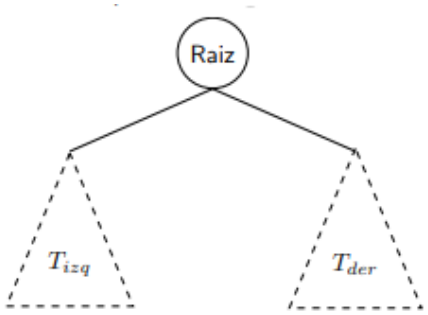


Figura 2: Árbol Binario.

Al igual que sucede con los árboles n-arios, no se puede establecer un método específico de insertar, depende del usuario la decisión de insertar un nodo a la derecha o a la izquierda de otro nodo (con la búsqueda y la eliminación sucederá lo mismo).

2.3. ABB

Un abb, o árbol binario de búsqueda, es un arbol binario con reglas para insertar nodos dentro del mismo. Cada elemento del árbol tendra un valor o clave, por lo que tendremos una forma de comparar los elementos para poder definir un **orden** dentro del árbol. Los elementos que tengan claves mayores se insertan en subárboles derechos y los de claves menores en los izquierdos. Cada subárbol es un abb.

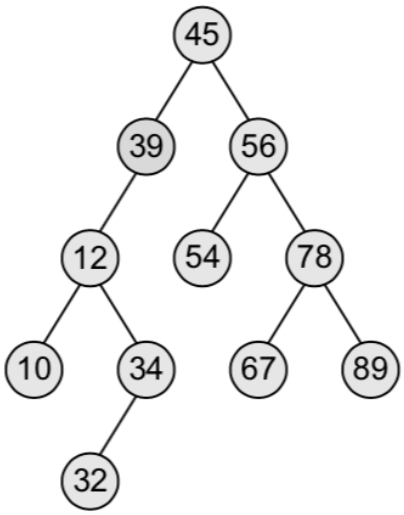
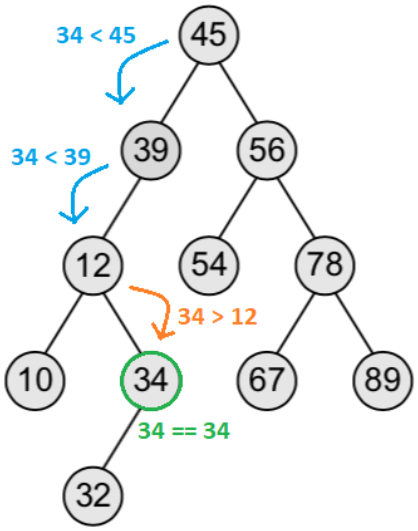


Figura 3: abb.

La **búsqueda** en un abb comienza en el nodo raíz y se siguen los siguientes pasos:
1. La clave que se busca se compara con la clave del nodo raíz.
2. Si las claves son iguales, la búsqueda se detiene.
3. Si la clave buscada es mayor a la clave del nodo raíz, se reanuda la búsqueda en el subárbol derecho.

3bis. Si la clave buscada es menor a la clave del nodo raíz, se reaunda la búsqueda en el subárbol izquierdo.

En la siguiente figura se muestra un ejemplo de búsqueda de un elemento con clave 34 dentro de un abb.



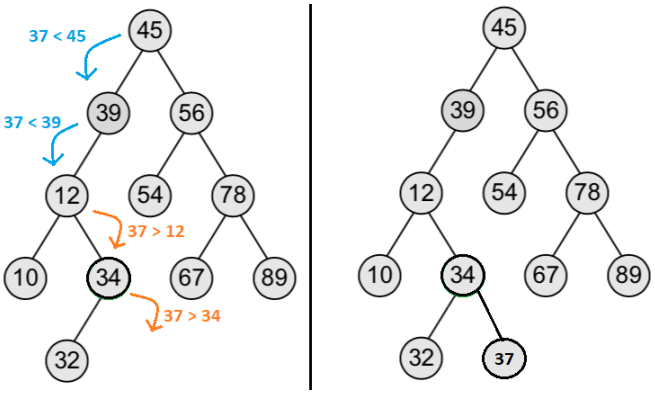
La **inserción** en un abb es similar, pero en este caso se busca hasta que un nodo no tiene hijo y se lo inserta en donde corresponda. Se siguen los siguientes pasos:

1. La clave que se busca insertar se compara con la clave del nodo raíz. Si es mayor, se avanza al subárbol derecho. Si es menor, al izquierdo.

2. Se repite el primer paso hasta que se llega al final de un subárbol. Si se encuentra una clave igual a la del elemento que se quiere insertar, hay dos opciones: dejar de avanzar y no insertar nada, para no tener elementos con claves repetidas; o seguir avanzanda hacia izquierda o derecha e insertarlo sin importar que esté repetida la clave. En la implementación del abb en este trabajo, se avanzó a la izquierda en este caso.

3. Se crea un nuevo nodo, al que se le asigna NULL al puntero izquierdo y al derecho y se lo coloca como hijo del anterior (izquierdo o derecho según cuál sea la clave).

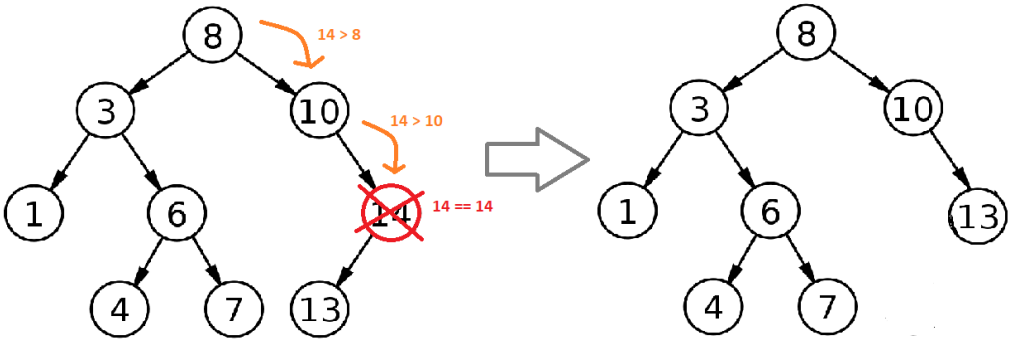
En la siguiente figura, se muestra cómo se avanza sobre un abb hasta que no en cierto momento un nodo no tiene hijo derecho. En ese momento, se crea un nuevo nodo con la clave que se quiere insertar y se lo pone como hijo derecho del último nodo sobre el que se comparó (el nuevo nodo no tiene hijo izquierdo ni derecho).



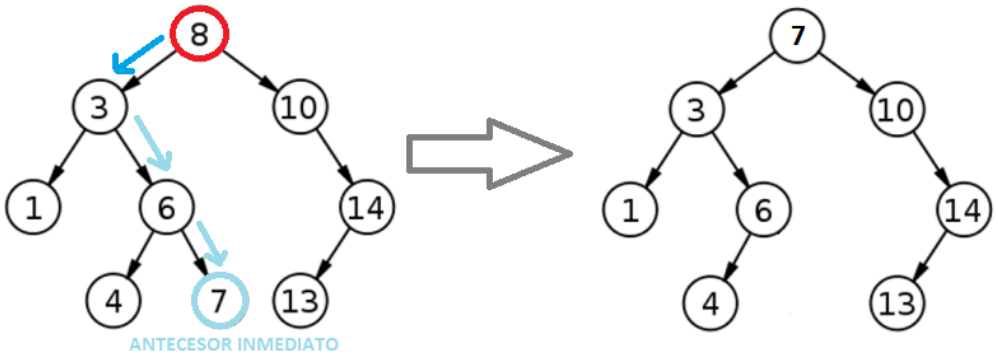
La **eliminación** en un abb se divide en distintos casos. En todos ellos, se busca el elemento a eliminar (si no se encuentra, no se elimina nada) y se procede:

1. Si el nodo a eliminar es un nodo hoja, se hace que el padre a NULL (si es hijo derecho, debe tener su puntero derecho como NULL y si es izquierdo se debe modificar el correspondiente) y se elimina el nodo. Por ejemplo, en la figura anterior, si se quiere eliminar el 37, se avanza en el árbol y cuando se lo encuentra, se hace que el 34 apunte a NULL a su derecha y se borra el nodo con el 37.

2. Si el nodo a eliminar tiene un solo hijo, se procede del mismo modo que en el caso 1, pero se hace que el padre apunte al hijo del que se elimina y se borra el correspondiente. Se ilustra en la siguiente figura:



3. El caso más complejo es cuando un nodo tiene dos hijos. Aquí, se busca el predecesor inmediato: al encontrar el nodo que se quiere eliminar, avanzamos por el hijo izquierdo inmediato para luego avanzar hacia la derecha del árbol, hasta que un nodo no tenga hijo derecho. Este nodo será el que reemplace al eliminado y tendrá como hijos a los que eran los hijos del que se eliminó. También se puede buscar el sucesor inmediato, que se encontrará moviéndose a la derecha para luego avanzar hacia la izuierda hasta que no haya un nodo. En este trabajo, se utilizó el primer método. Se ilustra en la siguiente imagen, en donde se elimina la raíz de un árbol:



3. Detalles de implementación

3.1. abb_insertar

Se utiliza una función recursiva para la inserción de elementos. Esta función recorre el árbol de manera recursiva e inserta el elemento correspondiente. Lo que se hace es empezar modificando la raíz del arbol (en el primer llamado a la función recursiva). Se chequea si la raíz no existe y, en caso de que sea cierto, se crea un nodo con el elemento a insertar y se devuelve este (quedando guardado como la raíz del árbol). Si el caso no es el anterior, se utiliza el comparador del abb para chequear si se debe insertar el elemento en el subárbol izquierdo o en el derecho. Si, por ejemplo, se debe insertar a la izquierda, se modifica el hijo izquierdo de la raíz con la función

recursiva. Se seguirá recorriendo el árbol de este modo hasta que se llegue a una posición NULL y ahí se devolverá un nuevo nodo creado (este se guardará como el hijo izquierdo o derecho del nodo en el que se estaba parando antes, porque es el último llamado recursivo que se hizo). Luego de insertarse el nuevo nodo, se "vuelve hacia atrás" en el árbol hasta devolver la raiz que se había pasado en el primer llamado.

3.2. abb_quitar

Nuevamente, se recorre el árbol de manera recursiva hasta encontrar el elemento que se quiere eliminar. Si el elemento no se encuentra (es decir, se llega a un nodo NULL) se devuelve NULL. El primer llamado se hace modificando la raiz del árbol con la función recursiva y se recorre hacia la izquierda o derecha utilizando el comparador del árbol, haciendo nuevos llamados recursivos para modificar el nodo izquierdo o derecho como se hacía en abb_insertar. Cuando se encuentra el elemento que se quiere eliminar, hay dos casos posibles:

El primero, cuando el elemento a eliminar tiene un hijo o ninguno. En este caso, se crea un nodo auxiliar llamado hijo_no_nulo en el que se guarda el hijo que no sea nulo (si ambos son nulos, esta variable es NULL), se libera la memoria del nodo correspondiente y se devuelve el no nulo, lo que hace que sea el hijo derecho o izquierdo del padre del nodo eliminado (determinado por el anterior llamado recursivo, en donde se modifico a izquierda o derecha según el comparador del abb).

El segundo, cuando el elemento a eliminar tiene dos hijos. Aquí, se busca el predecesor inmediato con una nueva función recursiva, buscando a partir del hijo izquierdo y recorriendo hacia la derecha de este, hasta que no haya ningun elemento a la derecha. Una vez que se encuentra, se devuelve este nodo lo que hace que sea el hijo derecho o izquierdo del padre del nodo eliminado, como era en el primer caso.

3.3. Iterador

Se utilizan tres funciones recursivas distintas, según el recorrido ue sea recibido por parámetro. Se explica el funcionamiento de una de ellas pues las otras dos son muy similares.

En el caso de la iteración inorden, se busca invocar la función primero sobre el subárbol izquierdo, luego en el nodo por el que se ingreso y finalmente se invoca sobre el subárbol derecho. Se ingresa al árbol por la raíz del mismo (si es NULL, se devuelve 0 directamente) y se inicializa en 0 una variable que será el contador de cuántas veces se invoca la función.

Se recorre hacia la izquierda y, cuando se llega al último nodo izquierdo, se invoca la función sobre este, se aumenta el contador en uno y se avanza hacia la derecha. Cuando se haya recorrido el subárbol derecho del nodo sobre el que estábamos, el contador tendrá el valor de uno (cuando se invoco sobre el elemento actual) más las veces que se iteró en el subárbol. Se devuelve el contador, lo que modifica el contador del padre. Se invoca la función sobre el padre, se recorre su subárbol derecho y se procede del mismo modo que antes, hasta que se llega nuevamente a la raíz de árbol, pero ahora el contador vale la cantidad de nodos que tenga su subárbol izquierdo, pues ya se iteró sobre todos los elementos de este. Se invoca la función sobre el árbol (aumenta una vez más el contador), se recorre el subárbol derecho y finalmente se devuelve el valor del contador, que será del mismo valor que la cantidad de elementos que tenía el árbol (se invocó la función sobre todos).

El relato anterior sucede siempre y cuando la función que recibe el iterador devuelva siempre true. En caso de que devuelva false al invocarse sobre un elemento, el contador aumenta en uno pero se guarda en una variable de tipo bool que ya no se debe seguir iterando, por lo que cada vez que se quiera avanzar a la derecha se devolverá directamente 0, o si se vuelve para atrás en el árbol ya no se iterará sobre el elemento en el que se está parado. Así, se vuelve a la raíz sin iterar sobre ningún elemento y el contador valdrá lo que valía cuando se iteró por última vez.

3.4. abb_recorrer

Para esta función se reutilizó el iterador interno. Se creo una estructura llamada arreglo y se utiliza una auxiliar de este tipo que guarda el array recibido, el tamaño del array (su tope) y la cantidad de elementos guardados en él (inicializada en 0). Se recorre el árbol con el iterador y la función que se le pasa es una que recibe el elemento del árbol sobre el que se está iterando y la estructura, la cual guarda el elemento en el array y chequea si este esta lleno: si no, devuelve true y se puede seguir iterando. Si se llenó, devuelve false para no seguir guardando elementos.