Informe del TDA ABB, Algoritmos y programación II curso-04 Méndez.

Alumno: Víctor Zacarías, Padrón: 107080.

Mail: vzacarias@fi.uba.ar

-Descripción: En este informe se incluyen explicaciones teóricas sobre el TDA ABB, así como explicaciones y diagramas que buscan hacer entender la implementación y el funcionamiento de cada una de las partes del mismo.

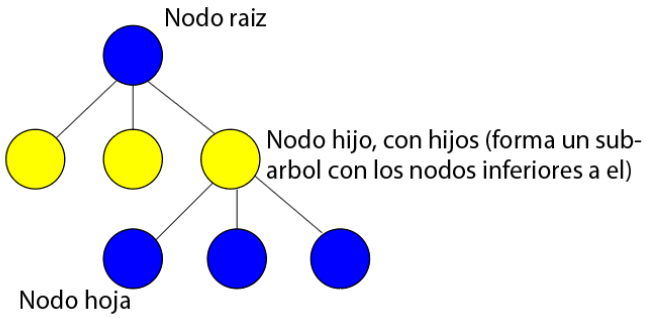
-Introducción teórica:

Un árbol es un tipo de dato abstracto que permite representar un conjunto de elementos a través de nodos dentro de una estructura que debe cumplir con las siguientes condiciones:

1-El primer elemento de un árbol se llama nodo raíz, y es el punto de entrada al mismo.

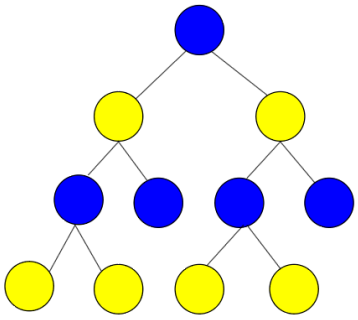
2-El nodo raíz puede estar conectado a 1 o más nodos (se les dice hijos), y a su vez estos pueden estar conectados a 1 o más nodos, y así sucesivamente.

3-Un nodo hijo que esté conectado a otros nodos, forma un subárbol dentro del árbol, a su vez este nodo, es la raíz del subárbol. Un nodo que no tenga hijos se llama nodo hoja.

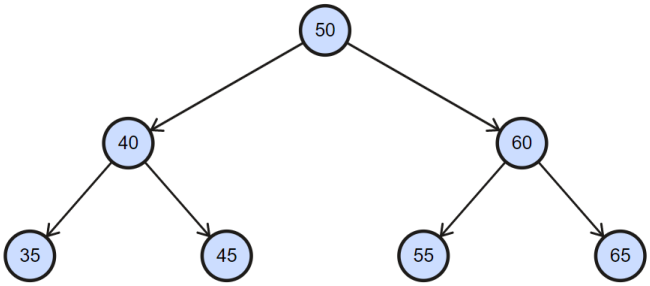


La necesidad de crear un árbol surge de representar una jerarquía en la representación de estructuras de datos, permite también optimizar la operación de búsqueda dentro de un conjunto de elementos. De esto último surgen distintos tipos de árboles, cada uno con una funcionalidad definida. Para la implementación del TDA ABB, se hacen uso del concepto de árbol binario y árbol binario de búsqueda (ABB).

Un árbol binario parte de la misma definición que la explicada para el árbol, con la particularidad de que cada nodo del árbol puede tener como mucho dos hijos.



A su vez un árbol binario de búsqueda parte de la misma definición de árbol binario, pero con la particularidad de que hay un criterio de orden al momento de insertar elementos dentro del árbol (Para el caso de este TDA se tomó la siguiente convención: Los elementos menores a cada nodo se insertan a la izquierda de estos, y los mayores se insertan a derecha). Esto último permite facilitar la búsqueda de elementos dentro del árbol (Al tener una noción de orden, es posible saber si buscar a derecha o izquierda un elemento).

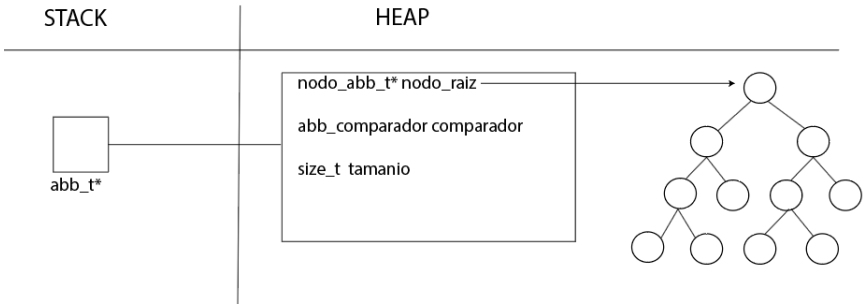


Nótese, que los elementos menores del árbol están a la izquierda de este, y los mayores a derecha.

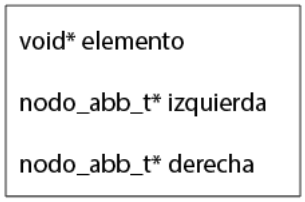
Es importante distinguir la diferencia entre cada uno de estos tipos de árboles, ya que cada uno cumple con una función determinada, y tiene distintos criterios al momento de insertar los elementos en el mismo.

- El nodo raíz de un árbol puede tener 1 o más hijos, y a su vez, estos pueden tener 1 o más hijos. No tiene ningún criterio al momento de insertarle elementos.
- En un árbol binario, los nodos pueden tener como mucho dos hijos, y no tiene ningún criterio al momento se insertarle elementos al mismo.
- En un árbol binario de búsqueda, los nodos pueden tener como mucho dos hijos, y si existe un criterio al momento de insertarle elementos (sirve específicamente para hacer operaciones de búsqueda).

-Detalles de implementación:



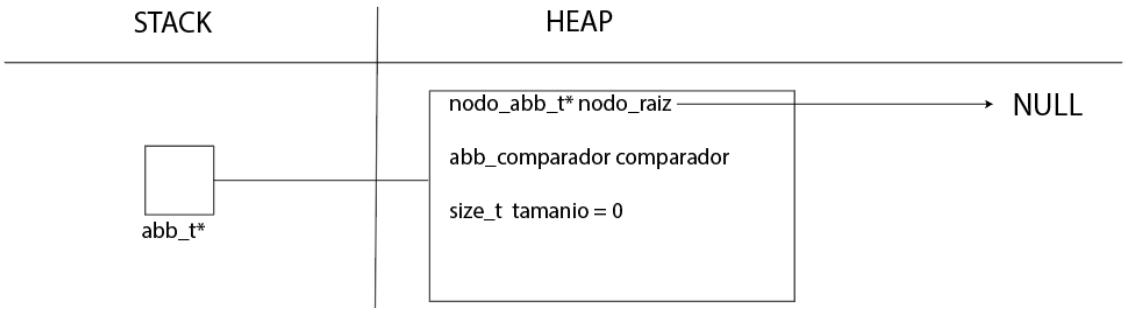
Estructura del árbol implementado en el TDA



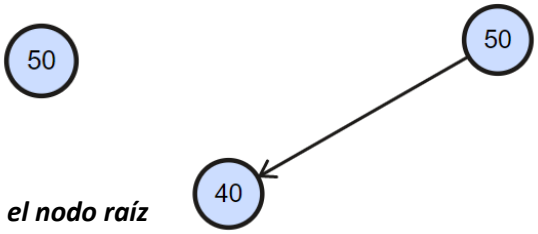
Estructura de un nodo

NOTA: Un nodo que no tenga nada a izquierda o derecha apunta a NULL.

1-Funcion abb_crear: Reserva memoria para crear un árbol vacío. Devuelve un puntero al árbol.

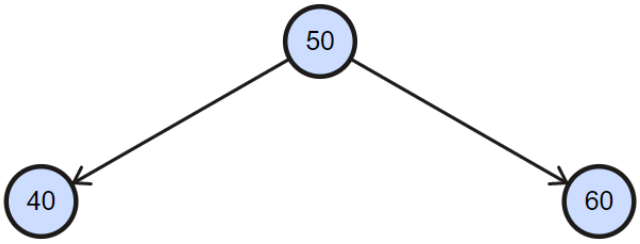


2-Funcion abb_insertar: Se encarga de insertar un elemento en el árbol, respetando la convención establecida para insertar elementos en este (menores a cada nodo a la izquierda, mayores a la derecha). Se tomo la convención de que, si el elemento a insertar es igual a otro, se inserta a la izquierda del que es igual.



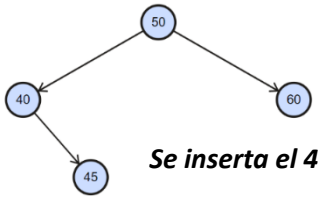
Se crea el nodo raíz

Se inserta un elemento menor al de la raíz

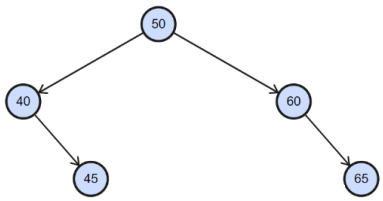


Se inserta un elemento mayor al del nodo raíz

Cada vez que se inserta un nodo en el árbol, se empieza comparando con el nodo raíz, luego dependiendo de si el nodo es menor o mayor, se va a izquierda o derecha, y así, se sigue comparando con los nodos que correspondan hasta llegar a una posición vacía (Se llega a una posición vacía cuando se llega a un NULL, acá es donde se inserta el nuevo nodo).



Se inserta el 45



Se inserta el 65

La función abb_insertar hace uso de una función auxiliar recursiva que con ayuda de un comparador (un criterio para comparar los elementos de un árbol), recorre el árbol a izquierda o derecha hasta llegar a una posición vacía, en donde se inserta el nuevo nodo. Esto último se hace a través de las llamadas recursivas (se llama a un nuevo nodo del árbol), cada vez que una llamada recursiva a un nodo termina, este se devuelve así mismo, y así sucesivamente. Con devolverse a sí mismo, se refiere a devolver la dirección de memoria que tiene un nodo.

Por ejemplo, al insertar el 45, 45 se devuelve a si mismo al 40, el 40 al 50, y el 50 (raíz) se devuelve a si mismo a la función que hizo uso de la función recursiva (abb_insertar). Esto último permite actualizar el árbol con la dirección del nuevo nodo. Finalmente, la función devuelve un puntero al árbol en el que se insertó nodo.

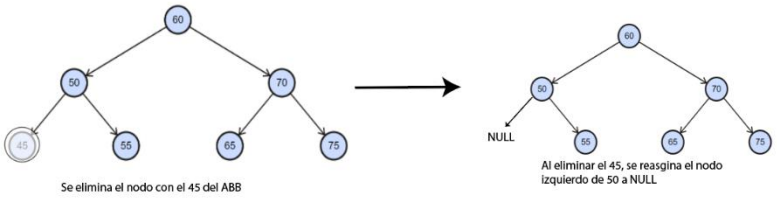
3-Funcion abb_quitar: Se encarga de eliminar un nodo del árbol, en caso de error o de que el nodo no exista, se devuelve NULL, en otro caso, se devuelve el elemento contenido en el nodo.

Existen diversos casos al momento de eliminar los nodos de un árbol, estos son:

- Se intenta eliminar un nodo sin hijos. (CASO 1).
- Se intentan eliminar un nodo con un hijo (CASO 2).
- Se intenta eliminar un nodo con dos hijos (Se adopto la convención de que el elemento a eliminar debe ser reemplazado por su predecesor inorden). (CASO 3).
    - El predecesor inorden no tiene hijo izquierdo. (CASO 3.1).
    - El predecesor inorden tiene hijo izquierdo. (CASO 3.2).
- Se intenta eliminar el nodo raíz (mismo caso que el anterior, con la particularidad de que se intenta eliminar la raíz del árbol). (CASO 4).

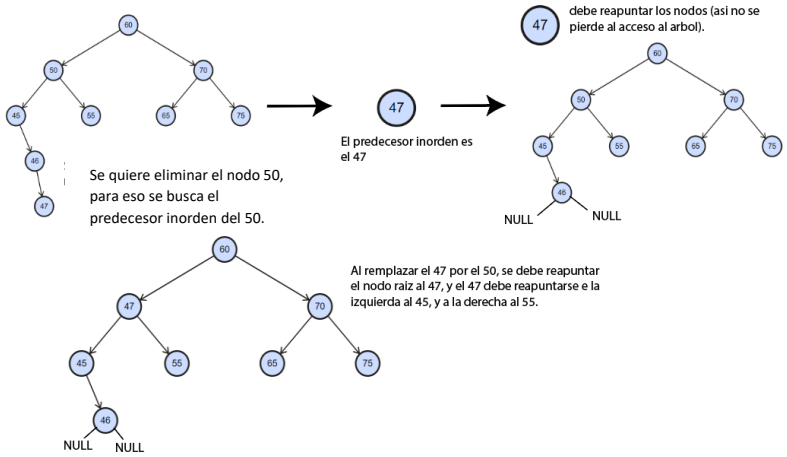
La funcion abb_quitar hace uso de una funcion auxiliar recursiva que permite recorrer el arbol hasta el elemento indicado para asi eliminar el nodo deseado. En esta funcion recursiva recibe por primera vez el nodo raiz del arbol (el arbol se recorre desde la raíz, recibe tambien el comparador (criterio con el que se comparan los elementos del nodo, quien es mayor o menor a otro), y una direccion de memoria a una variable booleana que busca determinar si se pudo eliminar el elemento o no.

Dentro de la funcion recursiva es donde ocurre todo el eliminar. Se recorre el arbol hasta encontrar el elemento eliminado, luego dependiendo del caso se elimina el nodo deseado.

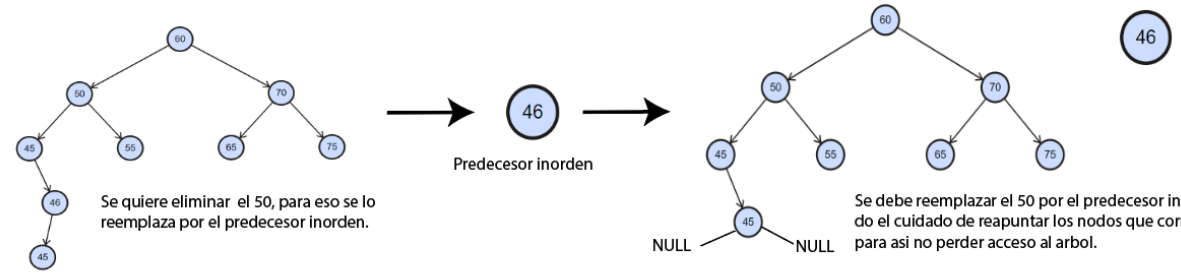


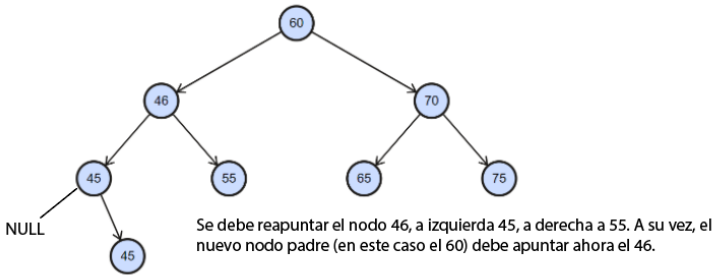
Caso 1: Se elimina un nodo sin hijos.

Caso 2: Se elimina un nodo con un hijo (se elimina el 54).



Caso 3.1: Se elimina con nodo con dos hijos, el predecesor inorden no tiene hijo izquierdo.



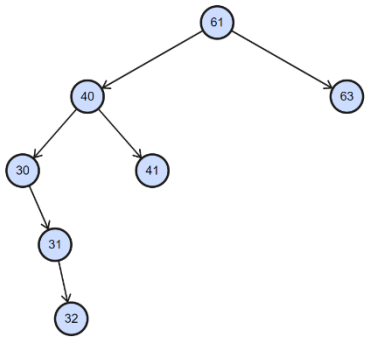


Como el predecesor inorden tenía un hijo a la izquierda, al reemplazar el elemento a eliminar por el predecesor inorden, se debe tener el cuidado de reapuntar el padre del predecesor inorden (45) al hijo del predecesor inorden. De lo contrario se perdería parte del árbol.

Caso 3.2: Eliminar nodo con dos hijos, el predecesor inorden tiene un hijo izquierdo.

En el caso en el que se intenta eliminar el nodo raíz, se puede aplicar los casos, 1, 2, 3.1, o 3.2, sobre el nodo raíz. Pero con el cuidado de que el nodo raíz es la entrada al árbol (véase, el abb_t tiene en uno de sus campos al nodo raíz), por lo que, si se elimina incorrectamente, se puede perder acceso a todo el árbol.

Se explico anteriormente que quitar hace uso de una funcion auxiliar recursiva para poder funcionar, y que en caso de que el nodo que se quiera eliminar tenga dos hijos, se debe reemplazar el nodo por su predecesor inorden. Dentro de la función auxiliar se hace uno de otra función auxiliar que se encarga de buscar el anterior al predecesor al orden del nodo a eliminar (se busca el anterior ya que, al mover de lugar al predecesor inorden, el anterior a el debe dejar de apuntar al predecesor inorden, ya que el mismo fue desplazado hacia otro lugar.



Si se quiere eliminar el 40 no se puede simplemente reemplazar por el 32 (predecesor inorden), es necesario saber quien es su anterior, ya que el cambiar de lugar al predecesor inorden, el 31 no puede seguir apuntando al 32. Esta fue la convención adoptada al momento de reemplazar por el predecesor inorden

4-Funcion abb_buscar: Se encarga de buscar si un elemento se encuentra o no en el árbol. Devuelve el elemento en caso de que se encuentre, o NULL en otro caso.

La función hace uso de una función auxiliar recursiva, quien recorre el árbol, comparando el elemento a buscar, con el elemento actual del árbol. Se empieza comparando con la raíz del árbol, dependiendo del resultado de la comparación se va a la derecha o la izquierda del árbol, y así hasta llegar a NULL, o al elemento buscado.

5-Funcion abb_destruir: Libera toda la memoria reservada para crear el árbol, es decir, cada uno de los nodos, y la estructura abb_t.

Para eliminar los nodos se usa una función auxiliar recursiva que hace un recorrido POSTORDEN del árbol, lo que permite recorrer el árbol, liberando los nodos de abajo hacia arriba.

6-Funcion abb_destruir_todo: Al igual que abb_destruir, libera toda la memoria del árbol, pero aplica un destructor sobre cada elemento de este (en abb_destruir se liberan los nodos y el árbol, pero no se hace nada sobre los elementos contenidos dentro del árbol).

Recorre también el árbol de forma POSTORDEN, cuando libera un nodo, le aplica un destructor. El destructor es una función del usuario, lo que hace depende de la implementación hecha por el usuario.

7-Funcion abb_recorrer: Se encarga de recorrer el árbol en 3 recorridos posibles (PREORDEN, INORDEN, POSTORDEN), que son determinados por el usuario, y de llenar un vector con elementos del árbol hasta una posición determinada (determinada por el usuario). Devuelve la cantidad de elementos que fueron almacenados exitosamente en el vector.

Para hacer esto se hace uso 3 funciones recursivas, se las detallara en forma general, ya que el comportamiento de cada una es muy similar.

```
guardar_en_array_inorden(arbol->nodo_raiz, array, tamanio_array, &cantidad_elementos_almacenados, &auxiliar);
```

Dependiendo del tipo de recorrido se usará una de las 3 funciones, en cada caso:

- nodo_raiz: Se envía el nodo_raiz ya que se busca recorrer al árbol desde la raíz.
- tamanio_array: Es el tamaño del vector que se quiere llenar.
- &cantidad_elementos_alamacenados: Debe ser una dirección de memoria de un size_t inicializado en 0, la idea es que lleve cuenta de cuantos elementos se han guardado exitosamente en el vector.
- &auxiliar: Cuando se lo manda a la función, deber ser la dirección de memoria de un size_t, inicializado en 0. La idea de usar punteros es que permite actualizar última posición del vector (la idea es que sirva como el i en las típicas iteraciones de un for).

Cada función recorre un árbol de una forma determinada hasta llenar el vector hasta donde haya sido indicado por el usuario, cada vez que se agrega un elemento, auxiliar se actualiza (es como el i en una iteración), cantidad de elementos almacenados se actualiza también (sirve para saber cuántos elementos fueron guardados exitosamente en el vector).

8-Funcion abb_con_cada_elemento: Recorre el árbol en un recorrido determinado (PREORDEN, INORDEN O POSTORDEN), y le aplica a cada elemento del árbol una función pasada por parámetro, abb_con_cada_elemento deja de ejecutarse cuando la función pasada por parámetro devuelve false. Al final se devuelven la cantidad de veces que la función pasada por parámetro fue usada.

La función dependiendo del caso (tipo de recorrido) hace uso de una función auxiliar recursiva (existe una para cada tipo de recorrido, son 3 en total). Cada una de estas funciones son muy parecidas, por lo que se detallara en forma general que hace una de ellas.

```
abb_con_cada_elemento_preorder(arbol->nodo_raiz, funcion, &cantidad, aux, &estado);
```

- nodo_raiz: Se debe mandar como primer parámetro el nodo_raiz, ya que desde él se empieza a recorrer el árbol.
- funcion: Es una función implementada por el usuario, es la función que se le aplica a cada uno de los elementos del árbol mientras sea posible.
- &cantidad: Es una dirección de memoria a un size_t, la idea de usar punteros, es que me permite mantener actualizada la variable cantidad, en ella guardo la veces que se le aplico funcion a un elemento del árbol.
- aux: es una variable proporcionada por el usuario para usarla en funcion.
- &estado: Es una dirección de memoria de una variable booleana que inicialmente es true. La idea de usar punteros es que mantiene actualizada el estado de la función, lo que permite cortar su ejecución cuando el estado pasa a ser false.

NOTA: Solo se agregaron descripciones sobre funciones cuya implementación no es simple, y de aquellas que necesitan un diagrama para que se entienda su comportamiento. Se agregaron también descripciones sobre aquellas funciones que hacen uso de funciones auxiliares, haciendo énfasis en las convenciones adoptadas en funciones como abb_con_cada_elemento (funciones recursivas auxiliares).