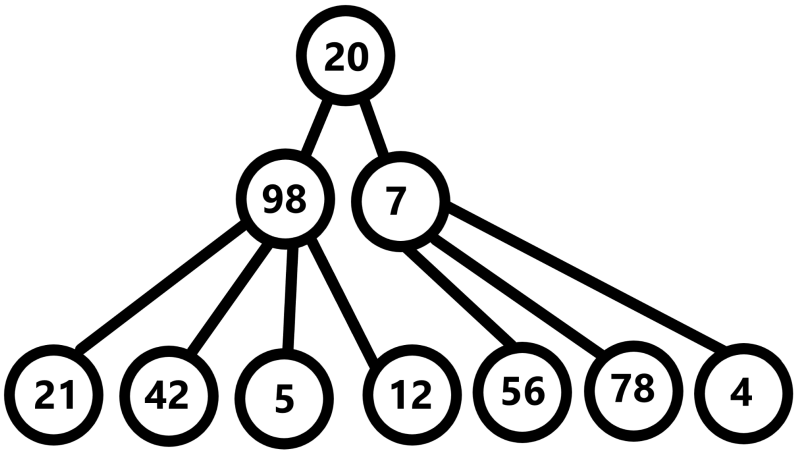
Marcos Bianchi - 108921

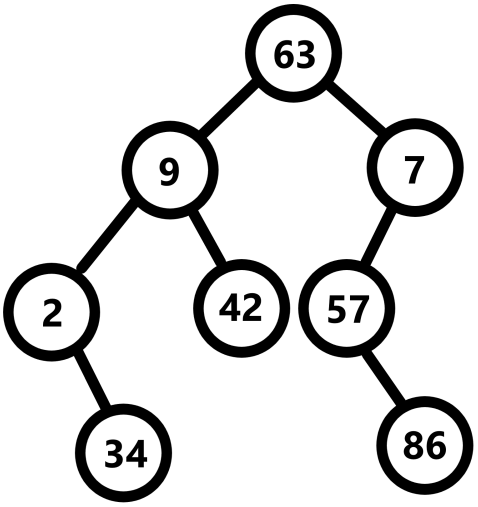
Árbol

Un árbol es una estructura de almacenamiento de datos compuesta de nodos en el que cada uno de ellos puede estar conectado a más de un nodo. En la estructura se encuentra un nodo principal (la raíz del árbol) que sería el punto de entrada al árbol. Para poder acceder a los datos dentro del árbol, uno va a tener que siempre arrancar buscando desde ella.



AB

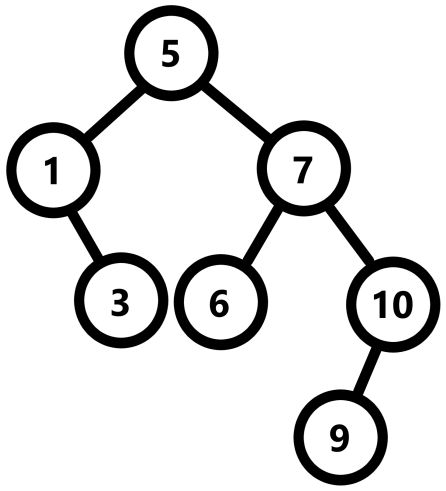
Un árbol binario es muy parecido a un árbol simple pero tiene una restricción, cada nodo puede tener hasta 2 hijos. Esto es clave ya que de no cumplir esta condición automáticamente deja de ser un árbol binario.



Marcos Bianchi - 108921

ABB

Un árbol binario de búsqueda es muy parecido a un árbol binario pero tiene otra restricción, el ABB necesita estar ordenado teniendo en cuenta algún tipo de criterio. En este caso el árbol almacena números y a medida que se van insertando números al árbol se irán posicionando a la izquierda los números más bajos y a la derecha los números más altos.



Diferencias

La principal diferencia entre los 3 tipos de árbol es que un ABB sirve para almacenar datos de manera ordenada en donde uno puede buscar los datos con mayor eficiencia que utilizando un AB por ejemplo. Un AB no sirve para búsqueda porque sus elementos no están ordenados y en un árbol n-eario los métodos de búsqueda son más complicados y creo que su utilización no resultaría en una reducción del uso de memoria ni eficiencia en complejidad de operaciones. Yo creo que el uso de un árbol n-eario se reserva para cuando no estamos reservando datos pero si estamos intentando resolver un problema con varias soluciones, por ejemplo el problema de backtracking del

Complejidad

La ventaja que tiene implementar un árbol en contraste a implementar una lista por ejemplo es que el árbol cuenta con un nivel de complejidad computacional más bajo dependiendo del tipo de árbol. En un ABB la complejidad de sus operaciones tienden a ser O(log(n)) si el árbol está balanceado. A medida que uno ingresa datos al árbol empieza a perder eficiencia y es ahí cuando la complejidad incrementa, en cambio en una lista sus operaciones son O(n).

Recorridos

Preorden: Visita el nodo actual primero, luego el izquierdo y por último el derecho (NID). Este recorrido sirve para guardar el árbol en un vector sin perder su jerarquía. Si uno quiere “copiar y pegar” el árbol se debe hacer con este recorrido.

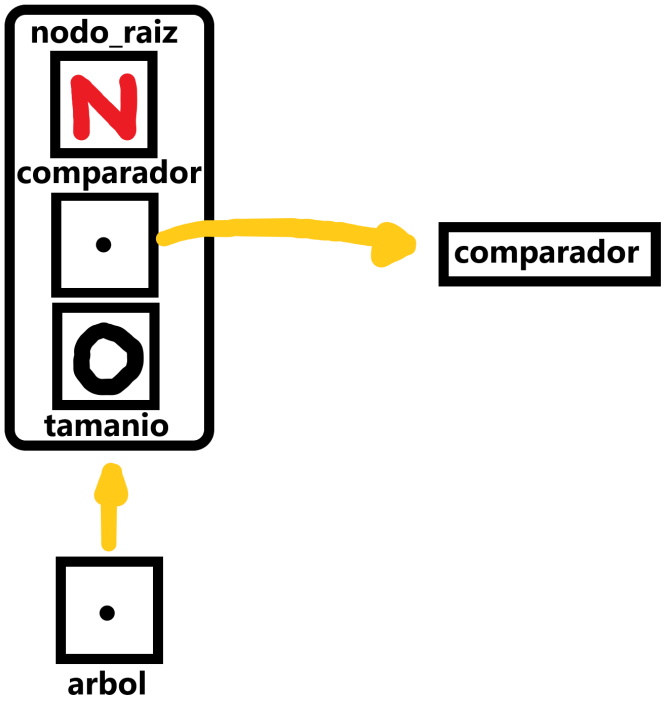
Inorden: Visita el nodo izquierdo, luego el actual y por último el derecho (IND). Este recorrido sirve para guardar el árbol de manera ascendente, los elementos más chicos del árbol serán leídos antes que los más grandes.

Postorden: Visita el nodo izquierdo, luego el derecho y por último el actual (IDN). Este recorrido sirve para eliminar un árbol ya que siempre que se visite el nodo actual será después de visitar todos sus hijos, esto quiere decir que el mismo no tiene hijos y no se van a perder las referencias a ellos ya que no existen.

Implementación

```c
abb_t *abb_crear(abb_comparador comparador);
```

abb_crear() reserva espacio en el heap para un abb_t usando calloc(), apunta el puntero comparador del árbol a la función pasada por parámetro y devuelve el puntero al árbol. En caso de que la función pasada por parámetro sea NULL la función devuelve NULL.



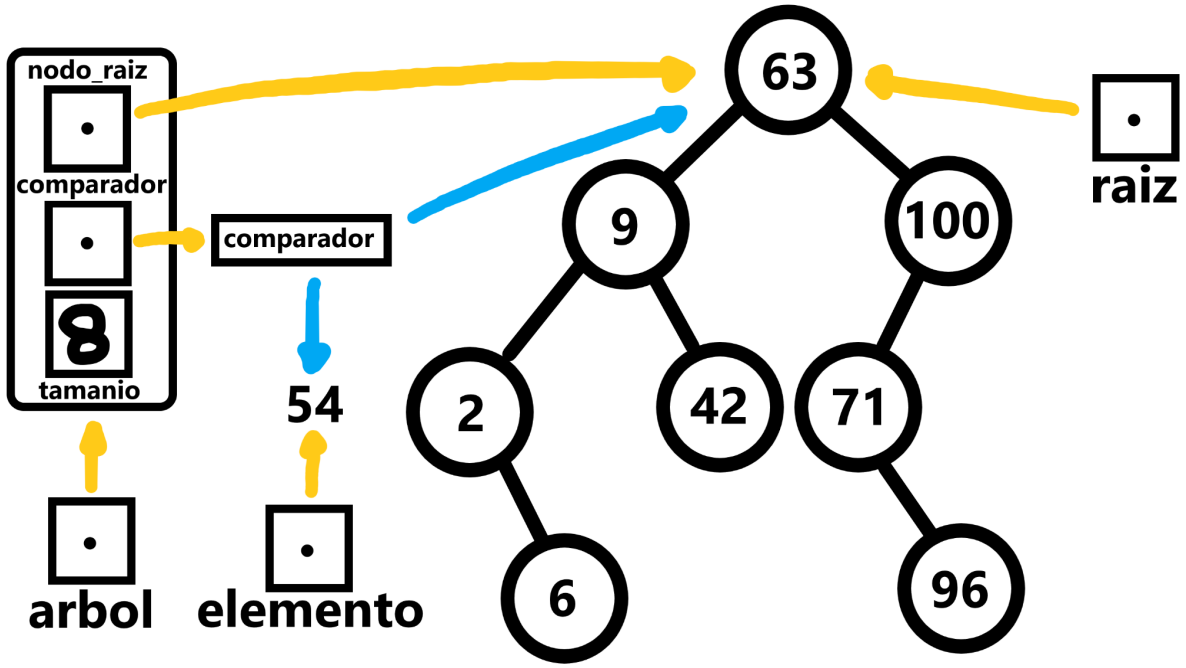
Este procedimiento tiene una complejidad de O(1).

```c
abb_t *abb_insertar(abb_t *arbol, void *elemento);
```

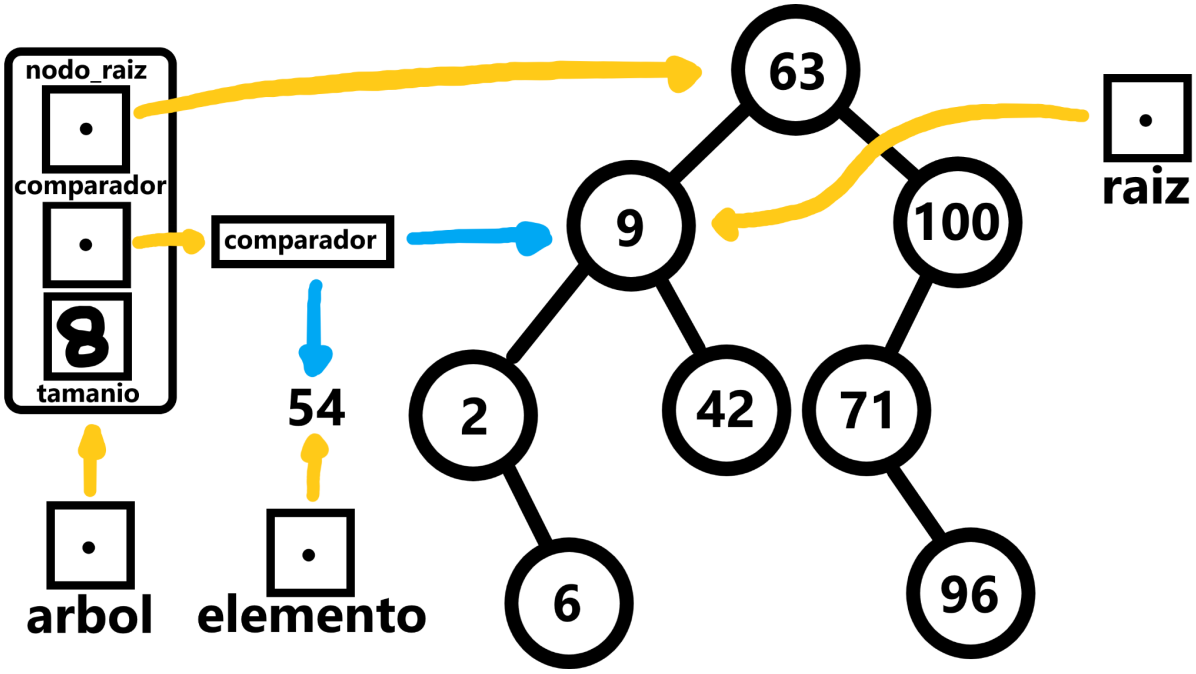
abb_insertar() primero verifica que el puntero al árbol no sea NULL, en ese caso devuelve NULL. Luego llama a una función auxiliar llamada insertar_hoja() a la cual le pasa el puntero

al árbol, un puntero al nodo_raiz y el puntero pasado por parámetro que apunta al elemento que queremos insertar en el ABB. Al finalizar el procedimiento de insertar_hoja(), apuntamos la raíz del árbol a la raíz devuelta por ella y devolvemos el árbol.

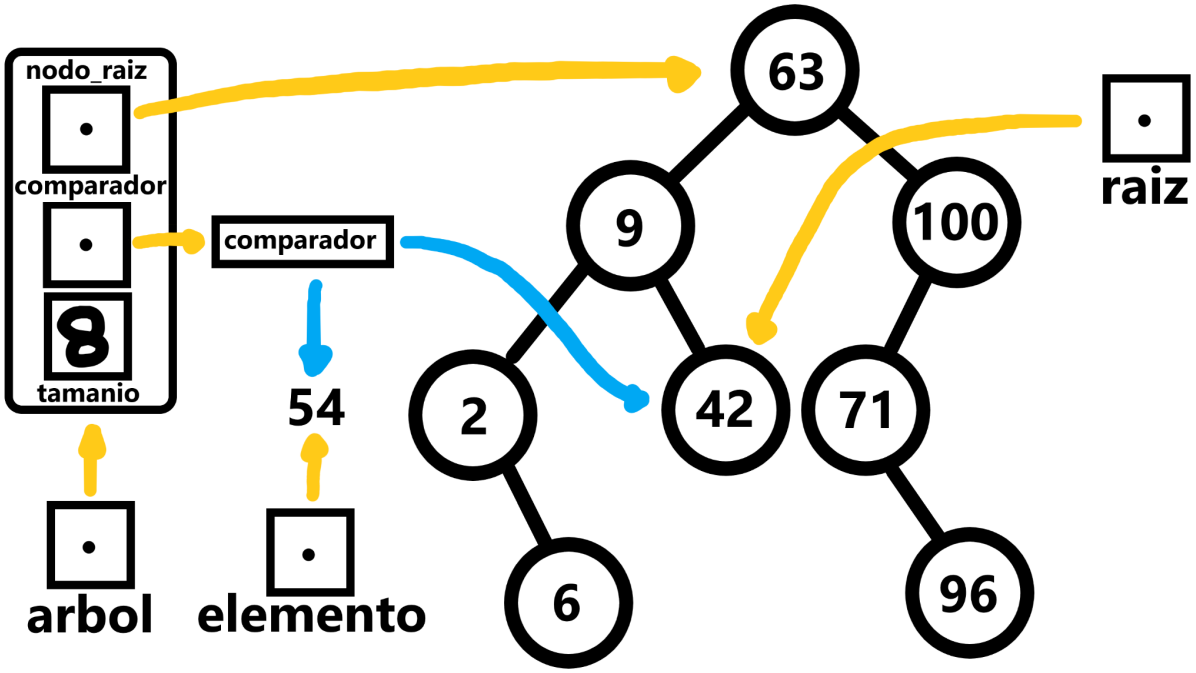
insertar_hoja() es una función recursiva que utiliza la función comparadora para navegar el ABB. Compara el elemento del nodo actual con el elemento que queremos insertar, dependiendo de lo que devuelva comparador() se hará 1 de 2 cosas, de ser más chico el elemento que tengo en la mano se visitará al hijo izquierdo y en caso contrario al hijo derecho. Este procedimiento se repetirá recursivamente hasta que nuestro puntero raiz esté apuntando a NULL.



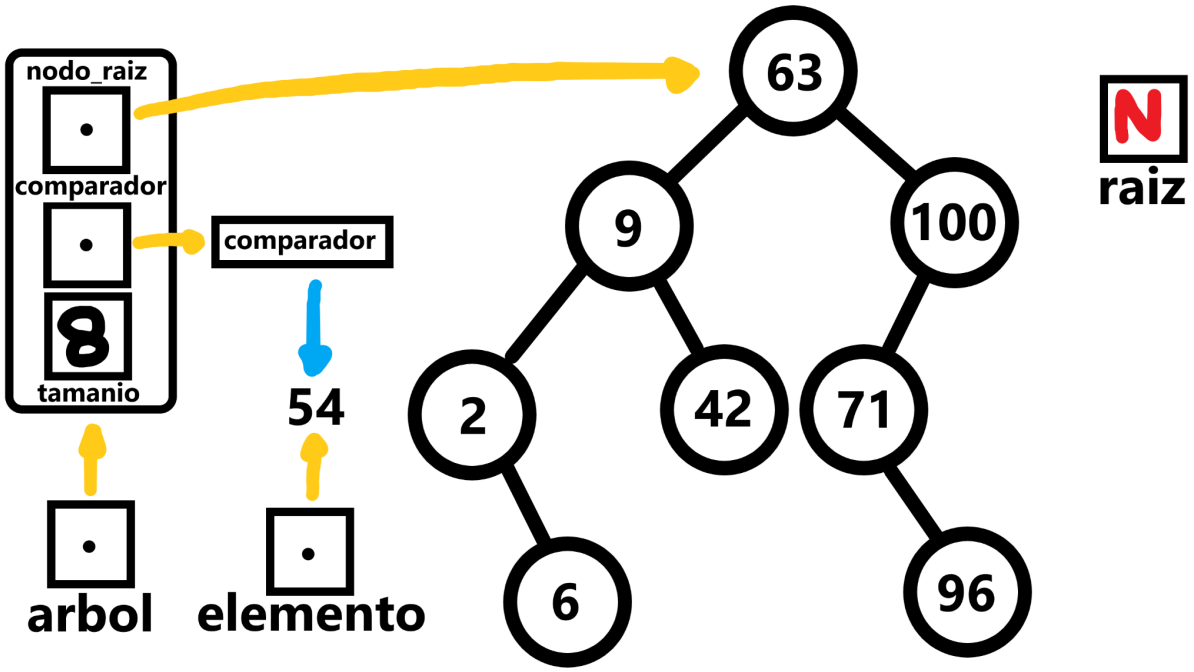
En este caso queremos insertar el número 54 en nuestro ABB. Primero comparamos el elemento al que apuntamos con el puntero raiz y lo comparamos con el elemento que queremos insertar. En este caso el comparador nos tira un número que determina que debemos visitar el hijo izquierdo porque nuestro elemento a insertar es más chico que el elemento actual.



Comparamos el nodo siguiente y el comparador determina que hay que visitar el nodo derecho.

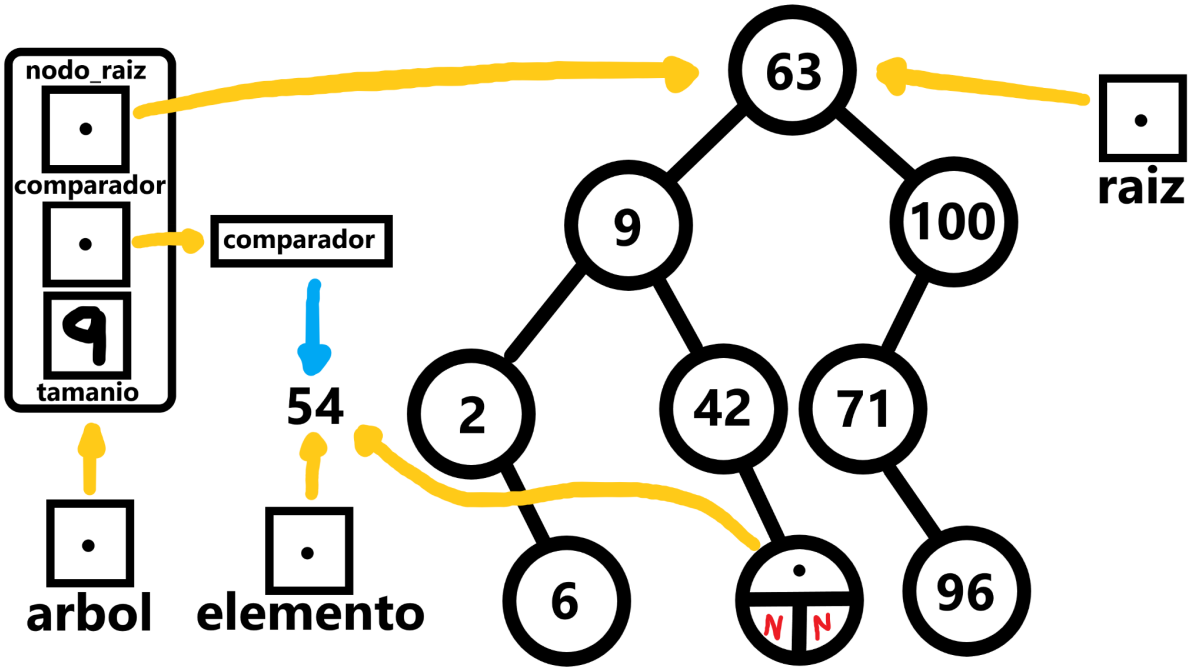


Otra vez comparamos el elemento actual y el elemento que queremos insertar y el comparador determina que hay que viajar al hijo derecho.



Ahora nuestro puntero raiz apunta a NULL ya que era a lo que apuntaba el puntero de hijo derecho.

Una vez que el puntero raiz apunte a NULL insertar_hoja() va a reservar memoria en el heap del tamaño de un nodo con calloc(), va a apuntar su puntero elemento al elemento que queríamos insertar desde un principio y va a incrementar en uno la variable tamanio del árbol.
Ahora la recursividad se encarga de “enganchar” este nuevo nodo al árbol devolviendo el nodo insertado y luego los nodos apuntados por el puntero raiz de cada iteración.

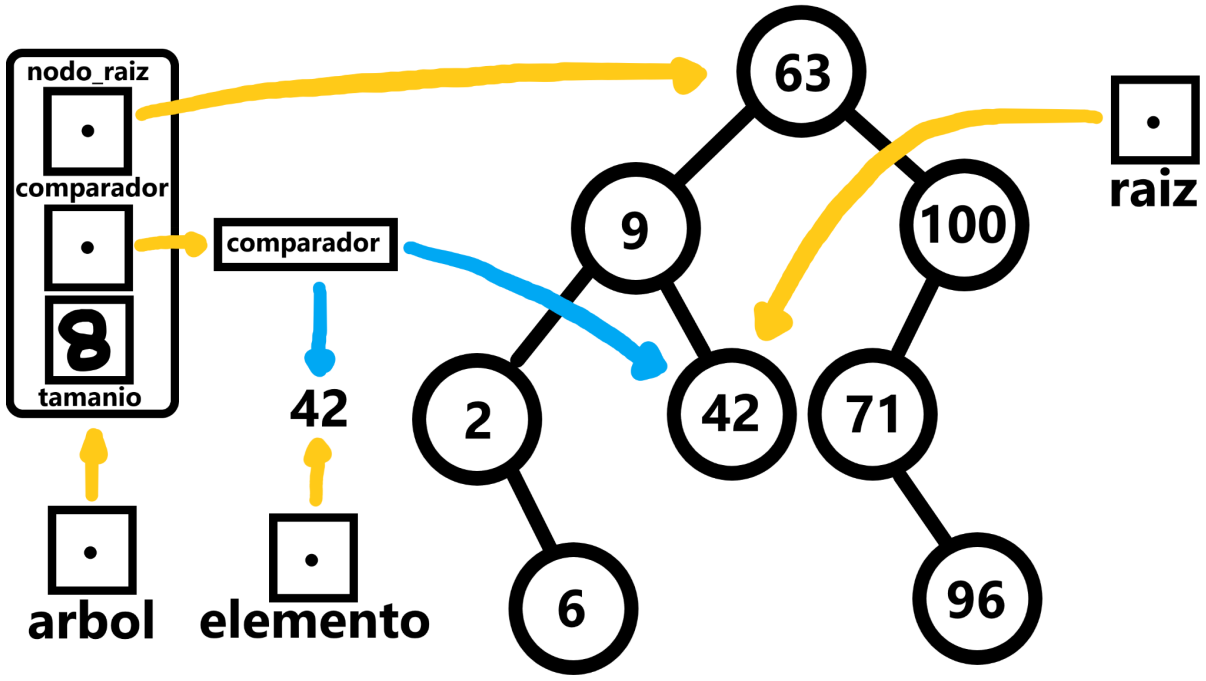


Este procedimiento tiende a tener una complejidad de O(log(n)). En el caso de que el árbol se haya transformado en una lista es de O(n).

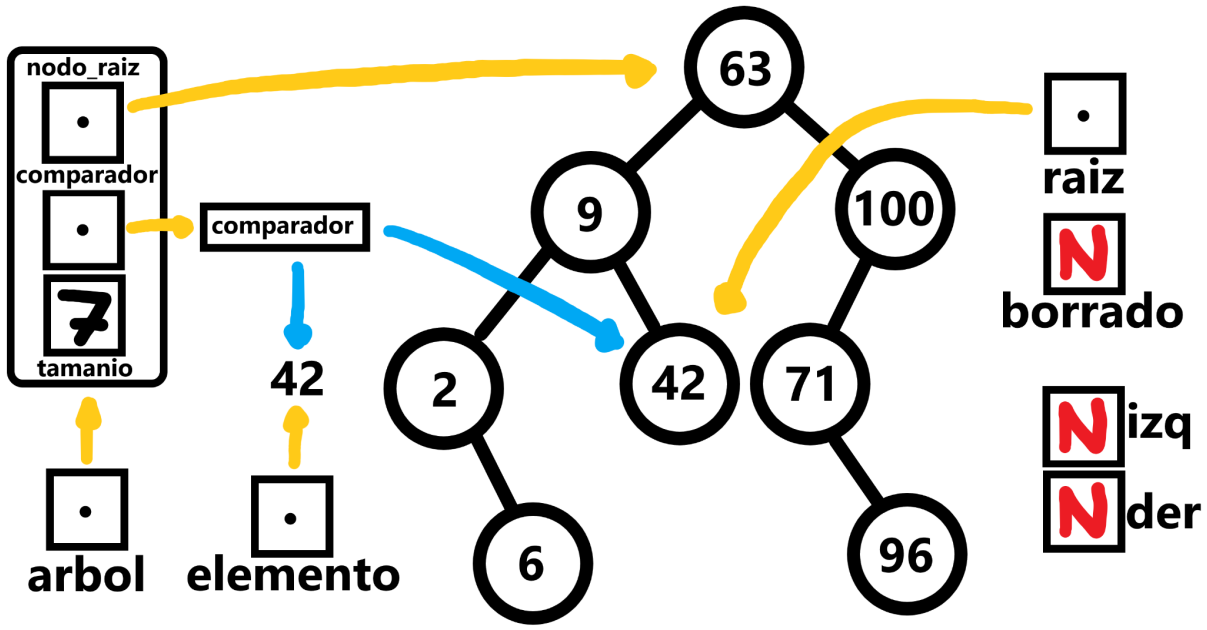
```
void *abb_quitar(abb_t *arbol, void *elemento);
```

abb_quitar() primero verifica que el puntero árbol no sea NULL, en ese caso devuelve NULL. Luego crea una variable de tipo void* que vendrá útil para apuntar con ella al elemento del cual nodo será borrado, después llama a una función auxiliar llamada quitar_nodo() que recibe por parámetro un puntero al árbol, un puntero a la raiz del arbol, un puntero al elemento que quiero borrar y la dirección de memoria del puntero que creamos hace unos instantes para poder modificar hacia donde apunta. Por último devuelve el puntero al elemento borrado.

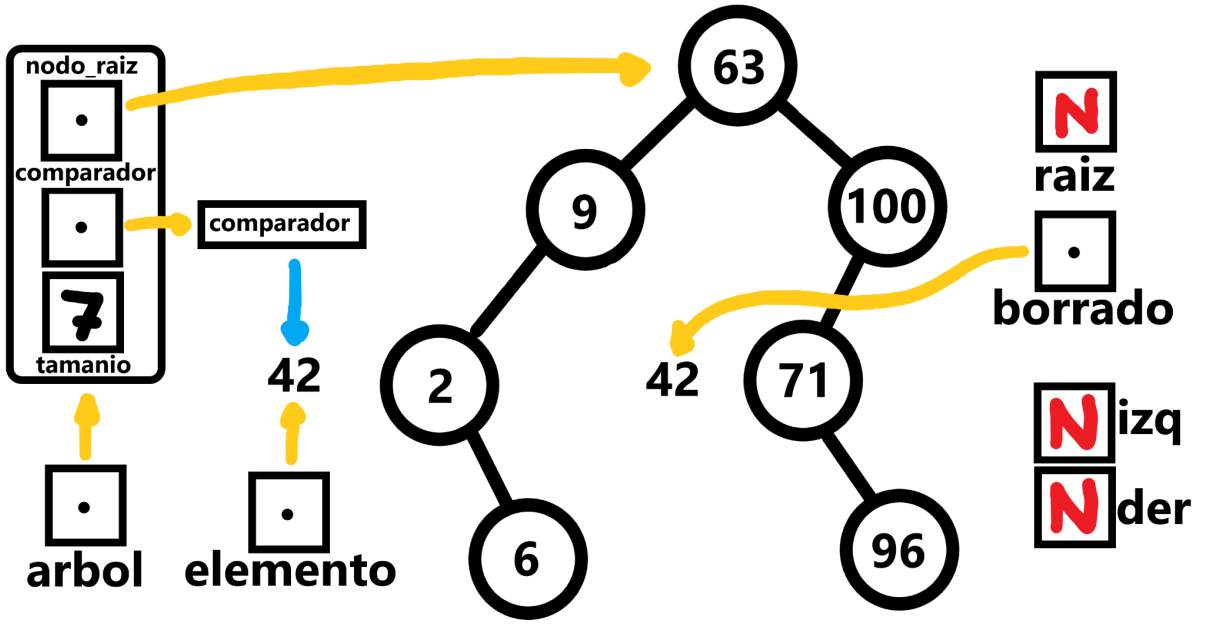
quitar_nodo() es una función recursiva que recorre el árbol en busca de un elemento que coincida con el elemento que le mandamos por parámetro para borrar su nodo. Recorre el ABB con la ayuda de la función de comparación que utilizamos antes para agregar un nodo pero con una diferencia, en caso de que la función devuelva que son iguales ahí es cuando quitar_nodo() determina que este es el nodo que queremos borrar.



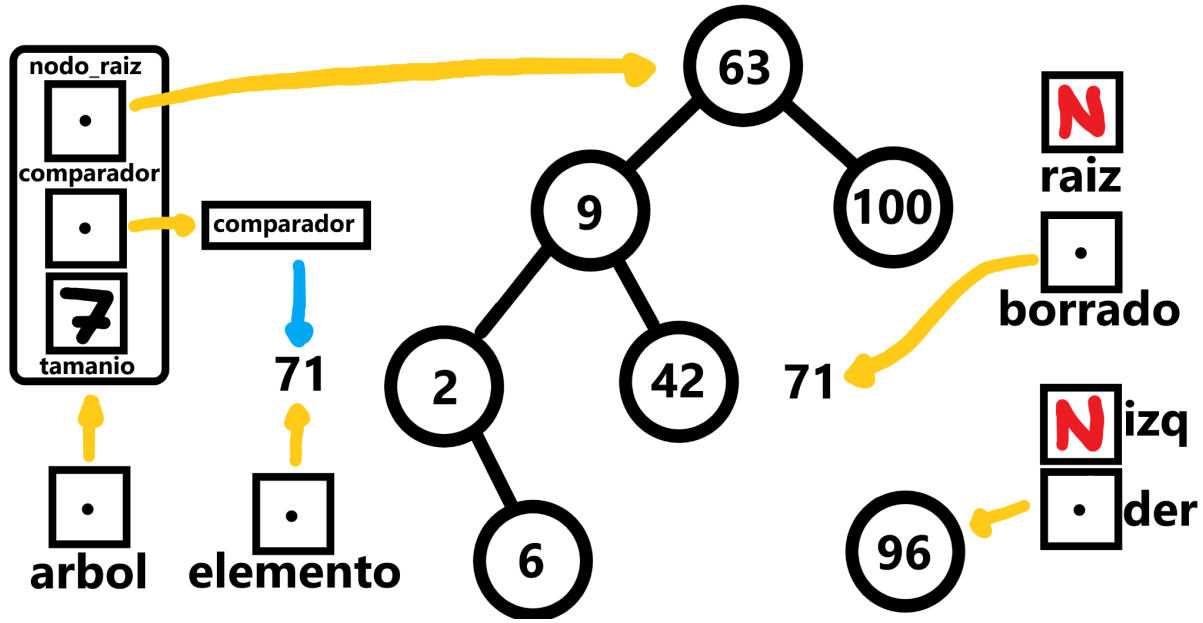
En este caso quitar_nodo() ya recorrió el árbol hasta encontrar un elemento que coincida con el elemento que queremos borrar. Primero vamos a apuntar al elemento con el puntero borrado que nos pasamos por parámetro, luego vamos a crear 2 punteros que apuntan hacia los respectivos hijos del nodo actual y vamos a disminuir el tamaño del árbol en 1.



Hasta acá este procedimiento se realizará siempre aunque el nodo a borrar tenga o no hijos. En caso de que no existan hijos se libera la memoria reservada por el nodo y se devuelve el valor actual del puntero izquierdo (NULL).

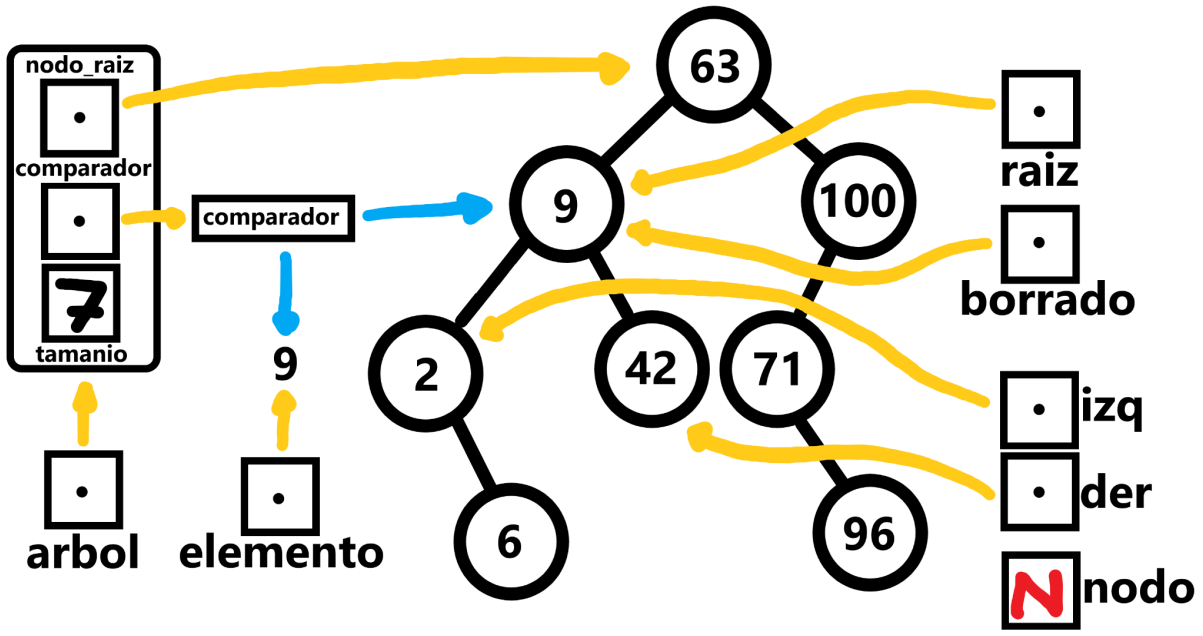


En caso de que el nodo tenga 1 hijo vamos a preguntar si el puntero izq es NULL, en caso de serlo vamos a devolver el valor apuntado por der.



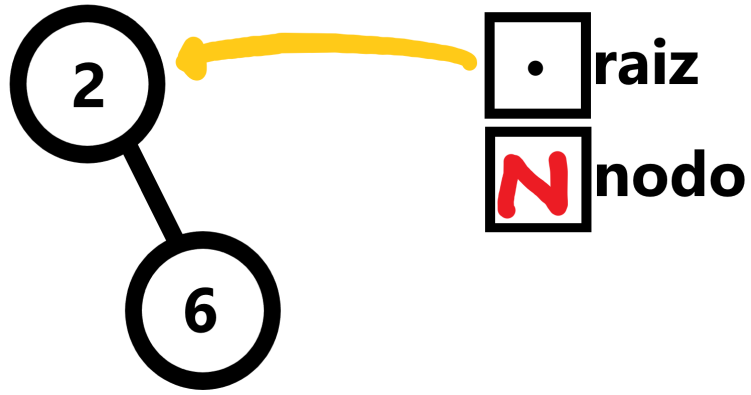
Al devolver el hijo derecho en este caso lo que estamos causando es que el nuevo padre (el 100) apunte al hijo del que borramos que sería el 96. También puede pasar que el 71 tuviera un hijo izquierdo y no uno derecho, quitar_nodo() se fija cuál de los dos no es NULL para devolver ese.

En el caso de que el nodo que queremos borrar tenga 2 hijos se va a crear una variable de tipo nodo_abb_t* llamada nodo apuntada a NULL y luego se va a llamar a una función llamada obtener_predecesor() que recibe por parámetro al hijo izquierdo y la dirección de memoria de la variable que declaramos recién. El resultado de la función se lo va a guardar la raíz como hijo izquierdo. Por último vamos a guardar el elemento del predecesor en donde apunta el puntero raiz y vamos a liberar la memoria utilizada por el nodo que queremos borrar devolviendo la raíz.

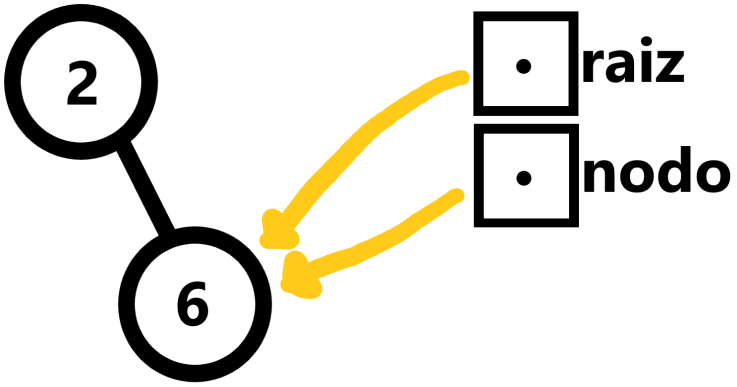


obtener_predecesor() va a recorrer el subárbol izquierdo hasta encontrar el predecesor que sería equivalente a ir todo por la derecha hasta encontrar el último nodo.

En la primer iteración el subárbol izquierdo debería verse así:

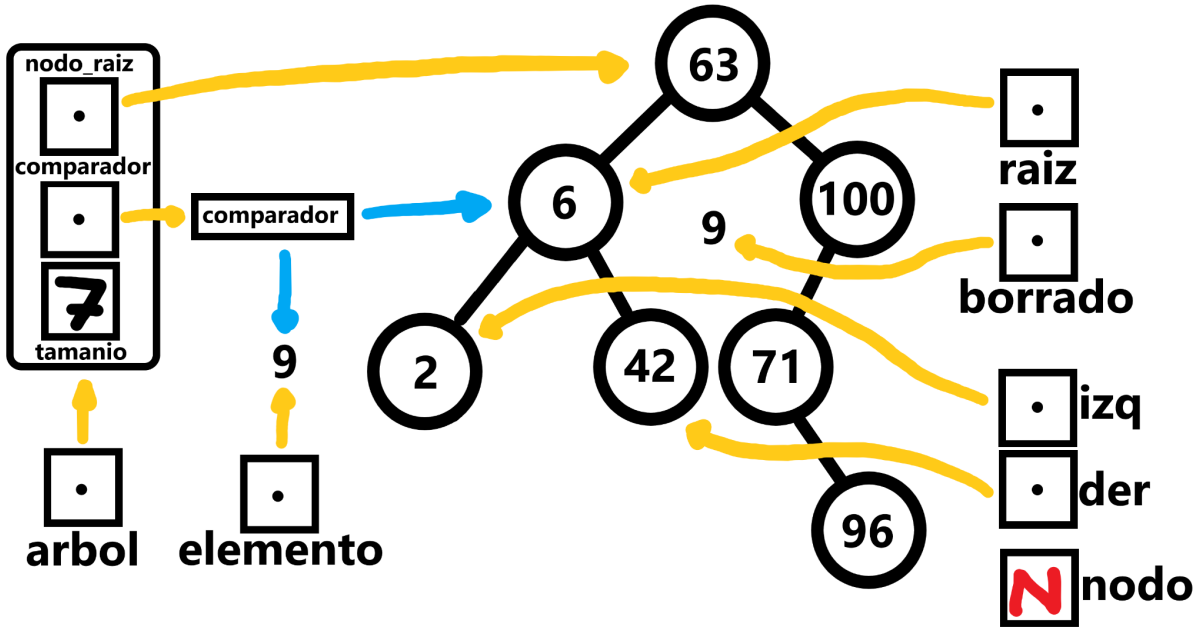


Todavía queda un nodo a la derecha así que vamos a visitarlo.



Aca la función deja de buscar otro nodo y se queda con el 6. Apunta nodo hacia donde apunta raíz y devuelve el hijo izquierdo para que sea luego apuntado por el padre (el 2). Esto va a arreglar un problema que podríamos tener en el caso de que el predecesor inorden tenga hijos, al devolver el hijo izquierdo arreglamos el problema ya que de tener un hijo devolvemos ese y de no tenerlos estamos devolviendo NULL por lo cual estamos desconectando el nodo del árbol.

Una vez que termina el procedimiento de obtener_predecesor() simplemente copiamos el valor guardado en el nodo del predecesor a la raíz, liberamos la memoria del nodo y devolvemos la raíz que guarda el nuevo valor.



Para finalizar la extracción de un nodo volvemos a abb_quitar() donde devolvemos el elemento apuntado por borrado.

Este procedimiento tiende a tener una complejidad de O(log(n)). En el caso de que el árbol se haya transformado en una lista es de O(n).

```
void *abb_buscar(abb_t *arbol, void *elemento);
```

abb_buscar primero verifica que el puntero árbol no sea NULL, en ese caso devuelve NULL. Luego verifica que el árbol no esté vacío y de no estarlo devuelve el resultado de una función auxiliar llamada buscar_elemento() que recibe como parámetro el puntero al árbol, un puntero al nodo_raiz y un puntero al elemento que estamos buscando.

buscar_elemento() es una función recursiva que busca en el ABB un elemento que coincida con el elemento pasado por parámetro. Utiliza la función de comparación del árbol para navegarlo y una vez que lo encuentra devuelve ese elemento. La función de comparación determina si buscar_elemento() debe buscar por el subárbol izquierdo o por el derecho, en caso de determinar que los elementos son iguales lo devuelve.

Este procedimiento tiende a tener una complejidad de O(log(n)). En el caso de que el árbol se haya transformado en una lista es de O(n).

```
bool abb_vacio(abb_t *arbol);
```

abb_vacio() es una función que verifica que el árbol esté vacío. Primero verifica si el puntero pasado por parámetro es NULL, de serlo devuelve NULL, luego si el tamaño del árbol es mayor a 0 devuelve false, de no serlo devuelve true.

Este procedimiento tiene una complejidad de O(1).

```c
size_t abb_tamanio(abb_t *arbol);
```

abb_tamanio es una función que devuelve el tamaño del árbol. Primero verifica que el puntero pasado por parámetro no sea NULL, de serlo devuelve 0. Luego devuelve el tamaño del árbol.

Este procedimiento tiene una complejidad de O(1).

```c
size_t abb_con_cada_elemento(abb_t *arbol, abb_recorrido recorrido,
                             bool (*funcion)(void *, void *), void
*aux);
```

abb_con_cada_elemento() es una función de iteración del árbol. Primero verifica que el puntero arbol y el puntero a la función pasada por parámetro no sean NULL, de serlo devuelve 0. Luego declara una variable llamada iterados que va a guardar la cantidad de nodos visitados por la función y una variable de tipo bool que dicta si se debe cortar la iteración o seguir navegando el ABB. Después llama a una función recursiva auxiliar llamada recorrer_abb_con_funcion() que recibe un puntero a la raíz del árbol, el tipo de recorrido, la función de corte, la variable aux (el contexto) y la dirección de memoria de iterados y exit. Por último devuelve el valor de iterados.

recorrer_abb_con_funcion() es una función que recorre el árbol de forma recursiva y utiliza un tipo de recorrido específico. Visita los nodos en el orden que declare el tipo de recorrido hasta que se recorra todo el árbol o la función pasada por parámetro devuelve false.

Este procedimiento tiene una complejidad de O(n).

```c
size_t abb_recorrer(abb_t *arbol, abb_recorrido recorrido, void
**array, size_t tamanio_array);
```

abb_recorrer() es una función de iteración del árbol. Primero verifica que el puntero arbol y el puntero al array no sean NULL, tambien verifica que el tamanio_array no sea 0, de ser válida alguna de estas cosas devuelve 0. Declara una variable de tipo int llamada iterados que guarda la cantidad de nodos visitados y llama a la función recursiva iterar_arbol() que recibe por parámetro un puntero a la raíz del árbol, el tipo de recorrido, el array, el tamaño del array y la dirección de memoria de la variable iterados. Por último devuelve el valor de iterados.

iterar_arbol() es una función que recorre el árbol de forma recursiva y utiliza un tipo de recorrido específico. A diferencia de recorrer_abb_con_funcion(), esta funcion guarda los elementos pisados en un vector y le suma 1 a la variable de iterados y deja de iterar el árbol una vez que se recorre todo el árbol o se haya llenado el vector con la cantidad especificada por tamanio_array en abb_recorrer().

Este procedimiento tiene una complejidad de O(n).

```c
void abb_destruir(abb_t *arbol);
```

abb_destruir solo llama a abb_destruir_todo() y le pasa el puntero arbol y NULL de función destructora.

```c
void abb_destruir_todo(abb_t *arbol, void (*destructor)(void *));
```

abb_destruir_todo() es una función destructora, libera la memoria reservada por el árbol y en caso de tener que liberar memoria que ocupen los elementos almacenados en ella requiere un puntero a función para ello. Primero verifica que el puntero arbol no sea NULL, de ser así termina su ejecución. Luego llama a la función auxiliar arbol_destruir() y le pasa un puntero al nodo_raiz y la función destructora.

arbol_destruir() es una función recursiva que libera la memoria ocupada por el árbol. Hace un recorrido postorden sobre el árbol liberando la memoria de los nodos, en caso de que la función recibida por parámetro no sea NULL la aplica en los elementos almacenados en el nodo al que apunta raiz, de ser NULL solo libera la memoria del nodo.

Este procedimiento tiene una complejidad de O(n).