Informe TDA ~ árbol binario de búsqueda

Padrón: 108951

Nombre: Luciano Andrés Lorenzo

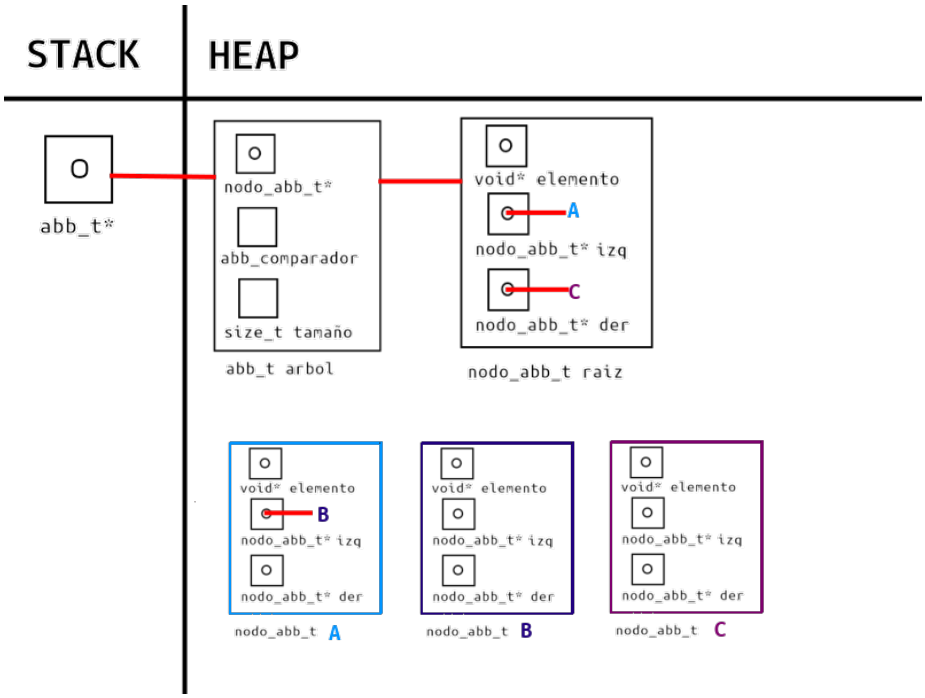
TDA ABB

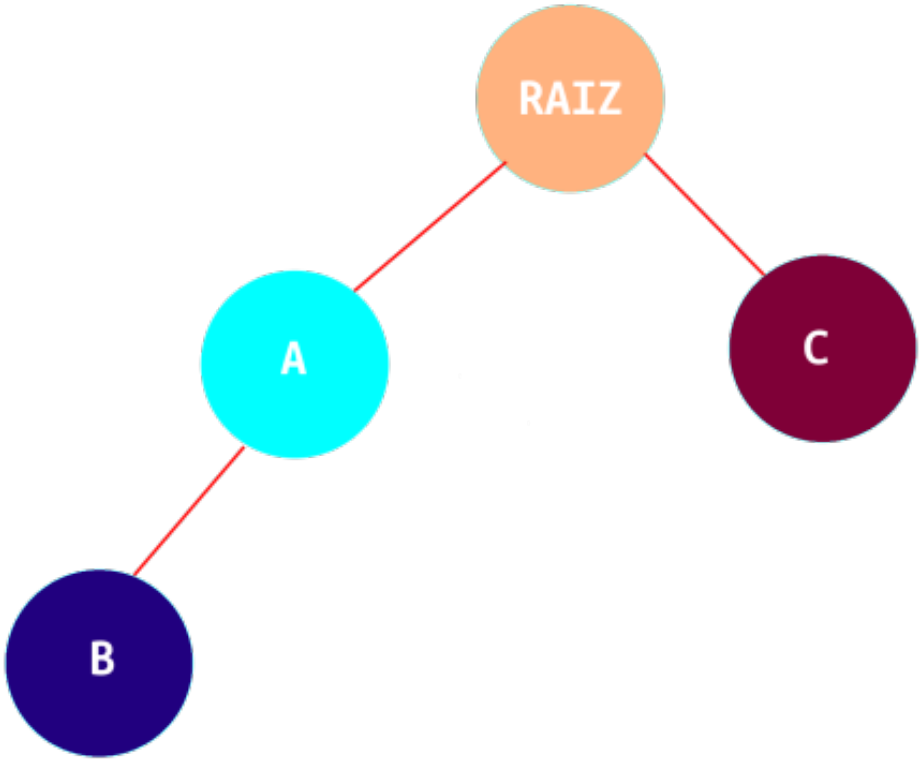
Un **árbol** es una estructura de datos que colecciona nodos, los cuales pueden estar conectados a múltiple nodos, siempre y cuando a cada nodo solo se pueda acceder desde un solo nodo (el padre) a excepción del nodo raíz que es el primero (el cual no tiene padre). Un **árbol binario** es un tipo especial de árbol, el cual solo puede conectar a 2 nodos (sus hijos). Y un **árbol binario de búsqueda** (*ABB*), es aquel que con un comparador siempre inserta los nodos mayores a la derecha y los menores (o igual en esta implementación) a la izquierda.

Al no ser lineal puede recorrerse de muchas formas.Como por ejemplo siempre empezar recorriendo los nodos izquierdos, luego los derechos y por ultimo el nodo de donde partimos (a este particular se lo llama *POSTORDEN*)

Al ser una estructura que se repite (cada nodo lo puedo pensar como su propia raíz de un subárbol), es muy **natural** implementar cada una de sus operaciones de forma **recursiva**. Por lo que mi primera decisión fue que cada función a implementar, llame ~por lo general~ a otra función pasándole por parámetro el primer del nodo del árbol (la raíz), y que esta se encargue de hacer la tarea recursivamente.

Todos los nodos y el struct árbol se guardan en el heap dinámicamente. En el mismo no tienen la forma de arbol como lo diagramamos normalmente, esta es solo una forma que tenemos para visualizarlos. Aca hay una comparación entre como se ve en la memoría un arbol y como su equivalente en una forma más comoda de visualizarlo.


STACK
HEAP
abb_t*
nodo_abb_t*
abb_comparador
size_t tamaño
abb_t arbol
void* elemento
A
nodo_abb_t* izq
C
nodo_abb_t* der
nodo_abb_t raiz
void* elemento
B
nodo_abb_t* izq
nodo_abb_t* der
nodo_abb_t A
void* elemento
nodo_abb_t* izq
nodo_abb_t* der
nodo_abb_t B
void* elemento
nodo_abb_t* izq
nodo_abb_t* der
nodo_abb_t C


RAIZ
A
C
B

La función más compleja a implementar fue `abb_quitar` en la cual hay que tener en cuenta muchos casos posibles a la hora de quitar un elemento, ya que al tratarse de un ABB, siempre que quito un elemento el árbol tiene que seguir siendo un ABB. Para poder devolver el elemento extraído, la parte recursiva de la función se va pasando por parámetro un puntero a puntero void, que cuando se encuentra el nodo que se quiere eliminar, es asignado al elemento del mismo. La función recursiva funciona reasignando el puntero al subárbol izquierdo o derecho con lo que devuelve la propia función recursiva en ese subárbol. Siempre se devuelve el mismo subárbol hasta que se encuentra un nodo a eliminar y se devuelve un reemplazo del mismo, este debe ser un nodo que se encuentre en algunos de los subárboles del nodo a eliminar. A veces es trivial determinar cual es el reemplazo, como por ejemplo cuando el nodo a eliminar solo tiene un hijo derecho, en ese caso ese mismo es el reemplazo. Pero puede pasar que tenga ambos hijos siendo ambos subárboles extensos, por lo que no solo hay que encontrar el reemplazo, sino que además tenemos que hacer que el reemplazo se conecte a los otros subárboles.

Para implementar la función `abb_recorrer` decidí reutilizar la función `abb_con_cada_elemento`. Como en esta hay que operar con el nodo y la cantidad de elementos, y a la función `abb_con_cada_elemento` solo se le puede pasar un solo auxiliar que pasa por parámetro cuando llama a la función con cada elemento, tuve que crear un struct interno que tenga el array, su tamaño máximo y un contador que lleve registro de la posición actual del último elemento insertado. De esta manera, paso por parámetro este auxiliar junto a la función que se encarga de recibirlo y agregar cada elemento a la lista. Puede que parezca un tanto confuso pero ahorra tener que escribir tres funciones diferentes para cada uno de los tres tipos de recorridos que puedo hacer (que ya habían sido implementados en `abb_con_cada_elemento`). Cabe aclarar que el struct implementado es solo interno a la implementación. Además que presenta punteros al array y a la posición actual del elemento final.

Operaciones

`abb_crear`: $O(1)$

- Se encarga de asignar memoria dinámica y devolver la referencia. Por lo que siempre tarda lo mismo.

`abb_vacio`: $O(1)$

- Comprueba si hay un nodo raíz, si no lo hay es porque el árbol está vacío. Por lo que siempre tarda lo mismo.

`abb_tamanio`

- Devuelve el tamaño que almacena el árbol en una variable, Por lo que siempre tarda lo mismo.

`abb_buscar`: $O(log(N))$

- La principal razón por la cual existen los arboles binarios de búsqueda. Se encarga de comparar recursivamente el elemento del nodo actual con el que quiero buscar. Si es mayor vuelvo a buscar a la derecha, si es menor vuelvo a buscar a la izquierda. Al estar dividiendo el problema cada vez en 2, es de complejidad logarítmica en base 2 (también se puede demostrar con el teorema del maestro).

`abb_insertar`: $O(log(N))$

- La complejidad se debe a la búsqueda que tengo que hacer hasta encontrar el nodo al cual le sigue el que quiero insertar, ya que la 'parte de insertar', es un término constante que no depende de la cantidad de elementos del árbol.

`abb_quitar`: $O(log(N))$

- La complejidad se debe a la búsqueda que tengo que hacer hasta encontrar el nodo que quiero eliminar, ya que la 'parte de eliminar', es un término constante que no depende de la cantidad de elementos del árbol.

`abb_destruir`, `abb_destruir_todo`: $O(N)$

- Tengo que recorrer y destruir los N nodos para destruir todo el árbol. Difieren en un término constante, la función destructor que se llama con cada elemento, pero que es constante.

`abb_con_cada_elemento`, `abb_recorrer`: $O(N)$

- Debe recorrer los N nodos en el peor de los casos, este ocurre cuando la función que llama con cada elemento nunca devuelve false (para dejar de recorrer). La complejidad es independiente del recorrido que se elija, ya que la cantidad de nodos a recorrer es la misma.