TDA ABB

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

Alumno:	Aldazabal, Lucas Rafael
Número de padrón:	107705
Email:	laldazabal@fi.uba.ar

Índice

1. Introducción

En este TDA se hace la implementación de un árbol binario de búsqueda convencional, con los hijos menores a la izquierda y mayores a la derecha. Se crean las operaciones de:

- Crear.
- Insertar.
- Buscar.
- Quitar.
- Recorrer a un vector (INORDEN, PREORDEN y POST ORDEN).
- Ejecutar una función a cada elemento (INORDEN, PREORDEN y POST ORDEN).
- Destruir.
- Destruir eliminando los elementos.
- y operaciones auxiliares como obtener tamaño y ver si está vacío.

2. Teoría

2.1. Árbol

Los arboles son un tipo de dato abstracto (TDA) que consiste en una estructura con características recursivas donde una colección de nodos se vinculan entre ellos donde cada nodo contiene un elemento y una lista de referencias a otros nodos inferiores (hijos), que tienen el mismo formato y de ahí la recursividad. A continuación se introducen los elementos que forman parte de un árbol.

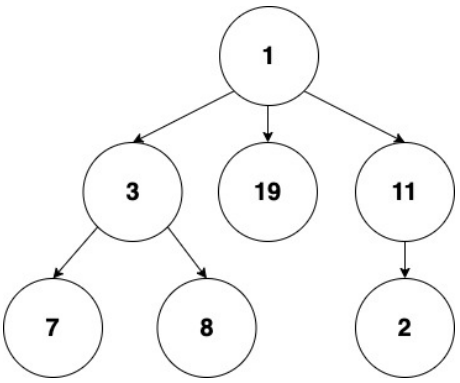


Figura 1: Ejemplo de árbol.

2.1.1. Nodo

Un nodo es un elemento básico de un árbol, es una estructura donde se almacena un valor y la referencia de otros nodo que parten de el (nodos hijo que lo tienen como nodo padre). Los nodos pueden ser internos (con al menos un hijo o externos (sin ningún hijo), también llamados nodos hoja.

2.1.2. Raíz

La raíz es el nodo padre de un árbol del cual salen los demás nodos, los cuales a su vez son nodo raíz de los subárboles que parten de el, de ahí la condición recursiva de la estructura.

2.1.3. Subárbol

Un subárbol es el árbol que se forma si a un nodo que a priori no es nodo raíz se lo abstrae de su árbol con su descendencia formando un nuevo árbol, esta abstracción no tiene porque ser real en el sentido de que no es necesario quitarlo o copiarlo sino que se simplemente cada nodo no hoja de un árbol es nodo raíz del subárbol que se crea con su descendencia.

2.1.4. Camino

Es la secuencia de nodos que hay que tomar (de padre a hijo recursivamente) para alcanzar un nodo.

2.1.5. Profundidad

La profundidad de un nodo es la cantidad de nodos que hay que recorrer para alcanzarlo desde un segundo elemento.

2.1.6. Relación entre Nodos

Cuando un nodo esta en el camino de un segundo nodo con la raíz del árbol, este primer nodo es ancestro del segundo y el segundo descendiente del primero. Si dos nodos tienen el mismo padre, estos nodos son hermanos.

2.2. Árbol Binario

Los arboles binarios son solo un tipo de arboles que tienen la limitante de tener como máximo dos hijos, introduciendo el concepto de hijo izquierdo y derecho. Los nodos pueden tener ambos hijos nulos siendo hoja, o tener un solo hijo (izquierdo o derecho), pero no mas. De esta forma se pasa de un vector de hijos a dos elementos fijos y permite que sus comportamientos a la hora de recorrerlos e interactuar sea mucho mas sencilla y predecible. Este tipo de arboles mas delimitados introducen operaciones como recorrer.

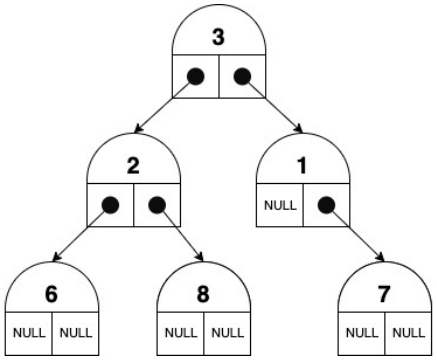


Figura 2: Ejemplo de árbol binario.

2.3. Árbol Binario de Búsqueda

Los arboles binarios de búsqueda (ABB) son una implementación de árbol binario donde se distingue al hijo izquierdo del derecho mediante una comparación (ejemplo en números cual es mayor), por estándar se suele utilizar los elementos menores en el hijo izquierdo y los mayores en el derecho. Esto aumenta aun mas la predictibilidad del árbol haciendo que buscar elementos en su interior sea altamente eficiente pudiendo partir de la raíz y mediante el comparador encontrar el camino a un nodo sin tener que probar los caminos posibles uno por uno hasta encontrarlo.

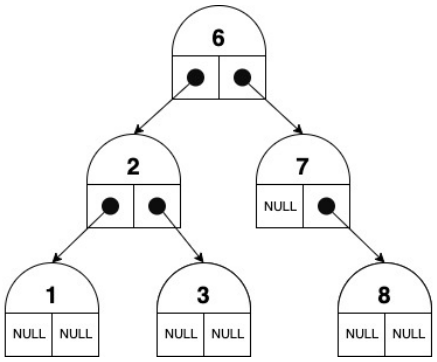


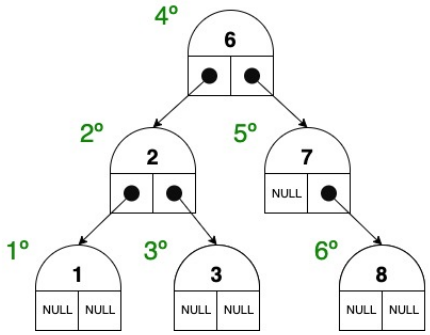
Figura 3: Ejemplo de árbol binario de búsqueda.

2.4. Recorrer árbol binario

A diferencia de otras estructuras como las listas que tienen una manera de recorrer, en los arboles no hay una relación lineal que marque un único camino a seguir, sino que existen seis posibilidades de recorridos recursivos con distintos enfoques y usos donde tres son los estándares: INORDEN, PREORDEN, POSTORDEN.

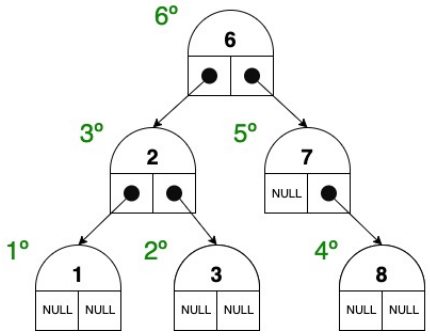
2.4.1. Recorrido INORDEN

El recorrido INORDEN consiste en de manera recursiva primero recorrer el subárbol izquierdo, luego de este terminado, recorrer el nodo raíz y luego el subárbol derecho, haciendo lo mismo en cada subárbol hasta alcanzar los nodos hoja. En el caso de los ABB los elementos terminan siendo recorridos en orden ya que siempre va primero el izquierdo (menor que la raíz), la raíz y luego el derecho (mayor que la raíz).



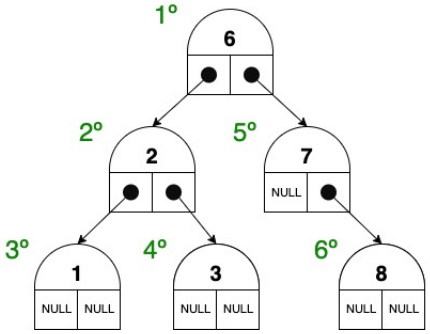
2.4.2. Recorrido POSTORDEN

El recorrido POSTORDEN tiene el mismo concepto que el INORDEN pero invirtiendo el orden, primero el subárbol izquierdo, luego el subárbol derecho y finalmente la raíz. La principal razón de este recorrido es para eliminar los nodos sin perder al padre primero y que quede aislado, y por ende inalcanzable.



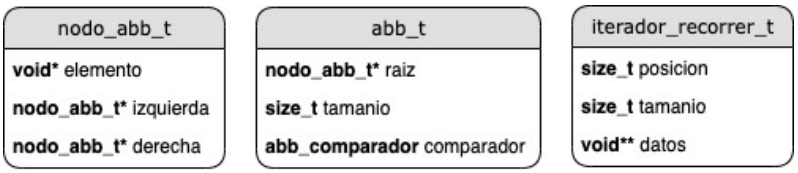
2.4.3. Recorrido PREORDEN

El recorrido PREORDEN es el contrario al POSTORDEN, primero recorre la raíz y luego el subárbol izquierdo y finalmente el derecho. Con esto se recorren los elementos respetando los nodos padre que sentaron las ramas y es la manera en la que hay que recorrerlo si se quiere duplicar el árbol ya que crea las ramas en el orden correcto para que no se altere.



3. Detalles de implementación

Para esta implementación de crearon tres estructuras de datos, los arboles, nodos y iterador para recorrer.



3.1. Crear

cuando se crea un nodo se reserva en el heap espacio para almacenar un abb_t y se lo inicializa con raíz nula, una función del usuario para comparar los elementos y tamanio cero, la función retorna el puntero a esta instancia, o nulo en caso de fallo reservando en el heap y/o recibe un comparador nulo .

3.2. Insertar

Cuando uno inserta un elemento en el árbol hay dos posibles comportamientos dependiendo del estado del árbol, si el árbol está vacío, o sea no tiene raíz, se crea el nodo con el elemento a insertar y se lo apunta en la raíz del árbol. Por otro lado, cuando el árbol no está vacío, se empieza

a buscar el lugar donde corresponde insertar el nuevo elemento desde el nodo raíz mediante el comparador del árbol, con este se decide si la inserción va a ser en el hijo derecho o izquierdo, una vez se define se pisa ese subárbol aplicando recursividad por una nueva inserción en este subárbol. Este proceso se repite hasta que el hijo donde se tiene que continuar la inserción es un subárbol nulo, en ese caso significa que se llego al lugar vacío donde se tiene que ubicar el nuevo elemento y se retorna el puntero del nodo creado que lo contiene, la recursividad en los returns actualizan los subárboles todo el camino devuelta al nodo raíz y con esto se tiene el árbol actualizado. Si la función falla se devuelve un NULL, en caso de éxito se devuelve el árbol actualizado (aunque al haber actualizado punteros no es necesario pisar el valor anterior).

3.3. Buscar

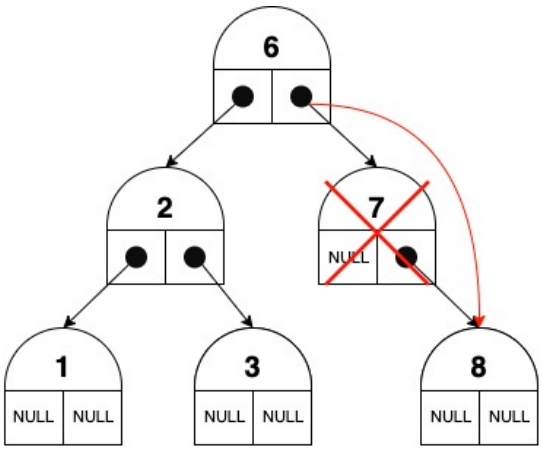
Al igual que la inserción, la implementación de búsqueda también es recursiva y consiste en, partiendo de la raíz, comparar el elemento buscado con el nodo actual y si es menor repetir con su subárbol izquierdo y si es mayor con su subárbol derecho. Esto continua hasta que el comparador determine que es el elemento buscado y se lo devuelve o que el nodo actual sea nulo lo que significa que el elemento no está en el árbol.

3.4. Quitar

A la hora de quitar, también se repite el concepto de insertar, partiendo de la raíz se compara el elemento a quitar con el nodo actual, si el elemento no es el deseado, se pisa el subárbol correspondiente con el otra llamada a quitar pero de este subárbol y así recursivamente hasta encontrar el elemento, acá arranca el proceso de quitado en el que hay tres alternativas: no tiene hijos, tiene un hijo o dos hijos.

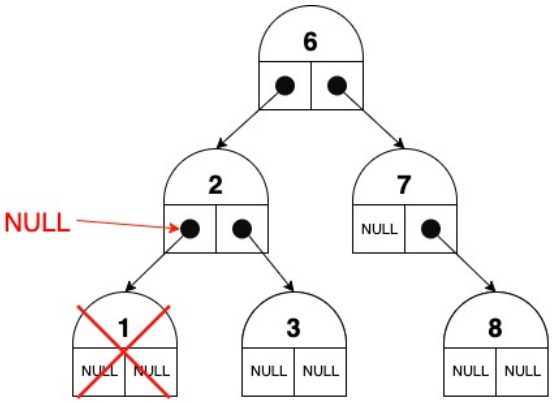
3.4.1. Un hijo

Cuando el nodo tiene un hijo lo de que se hace es guardar el puntero de este en un auxiliar para al eliminar el nodo deseado, reemplazar en el padre el hijo eliminado por su nieto. Esto resulta fácil debido a que como la implementación es recursiva retornando el subárbol para pisarlo simplemente devuelve el auxiliar luego de liberar el quitado.



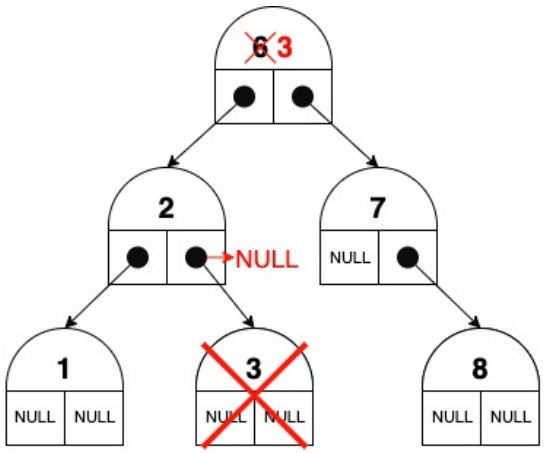
3.4.2. Ningún hijo

Esto es aun mas sencillo, simplemente se tiene que liberar el nodo y devolver nulo para que la rama de su padre que lo referenciaba no apunte al vacío. Lo que se puede hacer es incluir esta implementación en la de un hijo ya que cuando guarde en un auxiliar el hijo no nulo, va a tener nulo porque ambos hijos lo son y por ende va a retornar null siendo exactamente el comportamiento deseado.



3.4.3. Dos hijos

Cuando el elemento tiene dos hijos surge una complicación, es imposible conectar ambos hijos al padre del eliminado porque tendría dos hijos izquierdos (o derechos dependiendo de que lado sea el eliminado originalmente) e incluso tres en total. Para solucionar esto lo que se hace es buscar el elemento predecesor INORDEN, o sea el elemento que si ordenamos de mayor a menor es el anterior para reemplazarlo, con esto conseguimos que todos los demás elementos que se encuentran el subárbol queden bien ubicados igualmente ya que no hay ninguno que sea mayor a uno y menor al otro. Para encontrar este nodo predecesor hay un procedimiento que consiste en buscar el nodo mas a la derecha del subárbol que parte del hijo izquierdo del nodo a reemplazar, lo que asegura que si no hay mas elementos a la derecha, o sea mayores, del lado izquierdo del nodo, o sea menores, tenemos el mayor de los menores. Cuando tenemos este elemento simplemente se reutiliza el nodo del elemento a borrar para almacenar el elemento predecesor y tener los hijos y padre correctos. Existe la posibilidad que el predecesor tenga un hijo izquierdo no nulo que tenga que ser reacomodado antes de liberar este nodo que ya no se va a utilizar, en este caso lo que se hace es ponerlo como hijo derecho de su padre ya que si estaba como hijo del derecho es derecho al padre y se mantendrían perfectamente todas las condiciones, esto funciona en todos los casos menos una excepción, que el elemento que reemplaza no haya estado a la derecha del hijo izquierdo sino que sea el hijo izquierdo, en ese caso se posiciona como hijo izquierdo del nodo donde el eliminado estaba y ahora se encuentra el predecesor.



3.5. Recorrer elementos ejecutando una función

Otra de las operaciones mas importantes de un árbol binario es poder recorrer los elementos del árbol ejecutando una función por cada una que recibe el elemento como parámetro, acompañado de un auxiliar, para poder utilizar los datos almacenados. Alguna de las aplicaciones podría ser leer los datos para enlistarlos, sumar los números de un árbol, copiar el árbol a uno nuevo, guardar cada uno en una base de datos o cualquier otra cosa. Como se explicó en la teoría (sección 2.4), se puede recorrer un árbol en varias maneras y utilizamos las tres siguientes: INORDEN, PREORDEN y POSTORDEN. A la hora de utilizar los valores del árbol es importante elegir el orden correcto, por ejemplo si queremos copiarlo a otro debemos utilizar PREORDEN, si queremos borrarlo POSTORDEN, y si queremos recorrerlo en orden de menor a mayor INORDEN. En esta implementación, la función que se ejecuta para cada elemento retorna un bool que va a indicar si luego de ejecutarla para el elemento actual hay que seguir o se debe frenar acá (ejemplo, sumar valores menores a 6, se recorre INORDEN y cuando el elemento es mayor o igual no se suma y se devuelve false). Luego de recorrer elementos se retorna la cantidad de elementos a los que se les ejecuto la función (en nuestro ejemplo tendríamos la información de cuanto da la suma y cuantos menores a 6 hay (cantidad -1)).

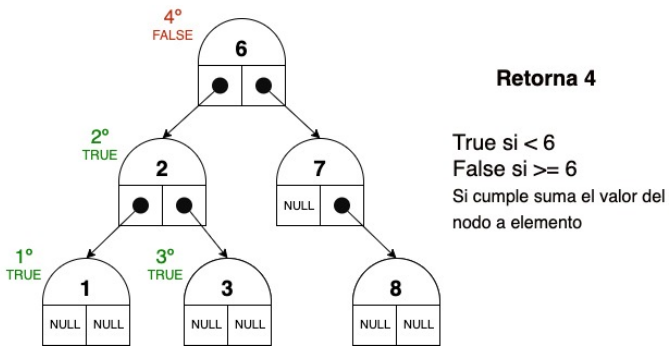


Figura 4: Ejemplo función a cada elemento.

3.6. Recorrer elementos a vector

Esta operación es una implementación prehecha de la operación anterior, consiste en dado un vector vacío y su tamaño se lo completa con los datos del árbol en el orden elegido, si el vector es mas grande que el árbol se guarda el árbol entero y se deja lo restante vacío, si es menor se lo completa y los elementos restantes no se guardan, luego la función devuelve la cantidad de elementos guardados. Para lograr esto se utiliza la estructura iterador_recorrer_t, mostrada anteriormente, guardando la posicion actual del vector (comenzando en 0), el tamaño del vector y el vector mismo, este iterador se utiliza como auxiliar de la función que se ejecuta a cada elemento la cual guarda en datos[posicion] el elemento actual e incrementa posicion, si posicion ya incrementado es igual a tamanio devuelve false porque esa posicion ya superaría el limite y debe frenarse ahí, caso contrario devuelve true.

3.7. Destruir y Destruir Todo

La ultima operación necesaria de un árbol es poder destruirlo para no perder memoria cuando se termina de ejecutar el programa. Para esto se recorre el árbol en POSTORDEN para, como se explica en la teoría, poder eliminar los nodos desde las hojas para no perder la referencia de algunos hijos todavía no liberados, esto se implementó con una recursividad simple ejecutando primero la misma función para el hijo izquierdo, luego el derecho y se libera el nodo actual, también esta la posibilidad de pasar una función del usuario que recibe el elemento por si quiere liberarlo en el momento a este también. Luego también se creo la función destruir que ejecuta esta primera sin pasarle el destructor y por ende dejando "vivos.ª los elementos de propiedad del usuario.

4. Otras operaciones implementadas

Se crearon dos funciones mas, una que devuelve el tamaño que tiene el árbol, es decir la cantidad de hijos que tiene actualmente y otra que devuelve un bool si el árbol esta vacío, es decir si tiene nodo raíz nulo o directamente es nulo el árbol.