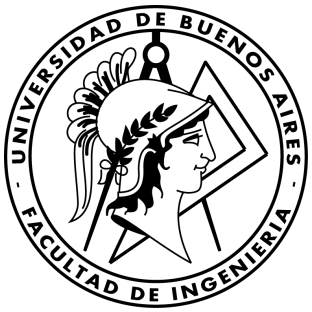

TDA - Arbol Binario de Búsqueda

[7541/9515] Algoritmos y Programación II
Segundo cuatrimestre de 2022

Alumno:	Bohórquez, Rubén
Número de padrón:	109442
Email:	rbohorquez@fi.uba.ar

Índice

1. Introducción

En esta ocasión se pidió implementar un árbol de búsqueda binario (abb). Este TDA consiste de un nodo principal llamado raíz y varios nodos subordinados que guardan un elemento y la referencia a dos nodos *hijos*. La estructura está ordenada de tal manera que uno de los hijos y toda su descendencia guardan elementos menores al nodo padre, y el otro los elementos mayores. Esta estructura es muy eficiente a la hora de buscar elementos, y por lo tanto es muy utilizada en diversas aplicaciones que requieren acceso eficiente a grandes cantidades de datos. Esta estructura también tiene una naturaleza recursiva, ya que se pueden pensar los hijos de un nodo como dos *subárboles*, por lo que es común que la implementación de las funciones relacionadas al abb sean implementadas de forma recursiva.

2. Teoría

Un árbol es una especie de generalización del concepto de lista enlazada, donde a cada elemento le pueden seguir más de un elemento. La estructura se asemeja entonces a un árbol (al revés), con un sólo elemento al inicio (la *raíz*) con varias ramas saliendo de ella, muy similar a los *diagramas de árbol* comunes en ciertas ramas de las matemáticas. Los elementos del árbol se les suelen llamar *nodos*, también es común usar la misma terminología que en un árbol genealógico para referirse a las relaciones entre nodos ("padre.^{es} el nodo anterior, "hijos los siguientes, etc.), y los nodos que no tienen hijos (o sea, que están *al fondo* del árbol) se les llaman *nodos hoja*. Los árboles también pueden ser pensados de forma recursiva. Cada uno de los nodos hijos de la raíz es, en el fondo, la raíz de un árbol también, que a su vez tiene como hijos más *subárboles*. En base a esta estructura se pueden definir varias cosas dependiendo de las necesidades del que lo utilice: *caminos* entre un nodo y otro recorriendo en cada paso uno de los hijos del nodo actual, *distancias* entre nodos, etc.

2.1. Árbol binario de búsqueda (ABB)

Un árbol binario de búsqueda es un tipo específico de árbol que, como su nombre lo indica, está construido especialmente para realizar operaciones de búsqueda. Es *binario* porque cada nodo tiene 2 hijos, que se les suelen llamar hijo izquierdo y derecho. Busca elementos aprovechando un orden o jerarquía entre ellos, por lo que también necesita una función que pueda comparar los elementos que se van a almacenar. Un abb tiene la siguientes características:

- Tiene un nodo raíz, que puede o no estar vacío.
- Tiene dos nodos hijos: uno a la izquierda y otro a la derecha, que también pueden o no estar vacíos.
- El hijo izquierdo es *menor* que el nodo raíz, y el hijo derecho es *mayor*.
- Ambos hijos también son abb's.

Si se cumplen estas condiciones, también se cumple una propiedad clave: *todos* los elementos a la izquierda de un elemento dado son *menores*, y todos los de la derecha son *mayores*.

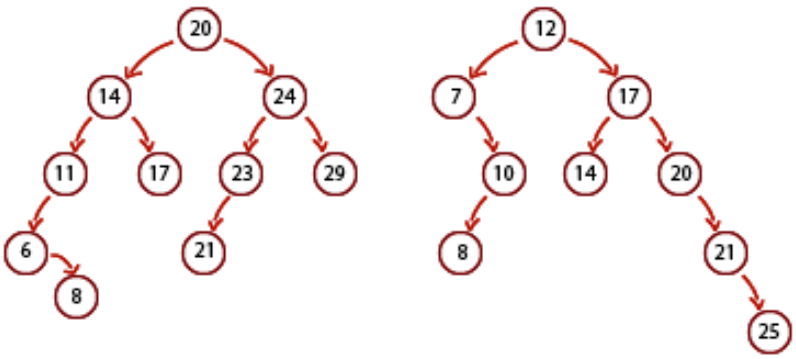

Figura 1: Árboles de búsqueda binarios.

Sabiendo esto, para buscar un elemento del árbol se puede comparar el elemento a buscar con la raíz, si es menor buscar a la izquierda o si es mayor buscar a la derecha, y repetir el proceso hasta que se encuentre el elemento o se llegue al final del árbol. Como al realizar esta comparación estamos eliminando la mitad del árbol a cada paso, la complejidad algorítmica promedio de buscar un elemento en un abb es de O(log(n)), que es una mejora considerable respecto a la complejidad O(n) que tendría buscar en una lista común cuando se tienen muchos elementos. La "forma"que tenga el árbol depende de en qué orden se inserten los elementos. Si se insertan en un orden desfavorable, el árbol puede llegar a parecerse mucho a una lista (ver el árbol derecho del diagrama más arriba), lo que impacta a la eficiencia de ciertos algoritmos del árbol (notablemente la búsqueda). Existen tipos de árboles *balanceados*, que mitigan este problema, pero no entran en el alcance de este reporte. Las operaciones básicas que suele tener un abb son las siguientes:

- **Crear/Destruir:** Se encargan de reservar memoria e inicializar, y liberar la memoria del abb respectivamente.
- **Buscar:** La operación característica del abb. Empezando por la raíz, compara el elemento del nodo actual con el elemento a buscar. Si el elemento a buscar es menor, continua la búsqueda por el hijo izquierdo, y si es mayor por el hijo derecho. Si son iguales, encontró el elemento y termina la función, y si llega a un elemento vacío, el elemento no está en el abb y también puede terminar la función.
- **Insertar:** Al insertar elementos, se debe tener cuidado de mantener la estructura del abb. Esto se logra insertando de una manera muy parecida a la que se buscan elementos, solo que, al llegar un elemento vacío, en vez de terminar la función se inserta el nuevo elemento en esa posición. Si el abb acepta elementos duplicados, se interpreta que el elemento a insertar sea igual al elemento de un nodo como si fuera mayor o menor (es indistinto siempre y cuando la decisión sea consistente).
- **Quitar:** Para eliminar un elemento del abb, primero se busca dentro del árbol. Si no está, la función no tiene que hacer nada, y si está, puede darse cualquiera de las situaciones siguientes:
  - Que el nodo a quitar *no tenga hijos*, en cuyo caso simplemente se puede destruir ese nodo (e indicar al padre de ese nodo que ya no existe).
  - Que el nodo a quitar *tenga un solo hijo*, en cuyo caso, antes de destruir el nodo, se tiene que poner a su hijo en el lugar que ocupaba ese elemento en el árbol para asegurar que no se pierdan elementos.

- Que el nodo a quitar *tenga dos hijos*, que es el caso más complicado. En ese caso lo que se hace es reemplazar el nodo a eliminar con el *antecesor o sucesor inmediato*. Si se quiere buscar, por ejemplo, al antecesor inmediato, se avanza al hijo izquierdo del nodo a eliminar y a partir de ahí se siguen recorriendo siempre por el hijo derecho hasta que se llegue a un nodo sin hijo derecho. Ese es el *mayor* de los elementos *menores* al elemento a eliminar (el sucesor sería el menor de los elementos mayores). Como sabemos que tiene como mucho un hijo (el izquierdo), puede dejarlo en su lugar en el árbol y pasar a reemplazar el elemento a eliminar. Esto garantiza que el árbol mantenga su orden, ya que buscamos el *mayor* de todos los elementos del *subárbol* de la izquierda, y como estaba a la izquierda sabemos que es menor que todos los elementos del *subárbol* de la derecha. Si se desea usar el sucesor inmediato en lugar del antecesor, el proceso es análogo solo que intercambiando izquierda por derecha y viceversa. Se explica a detalle más adelante.

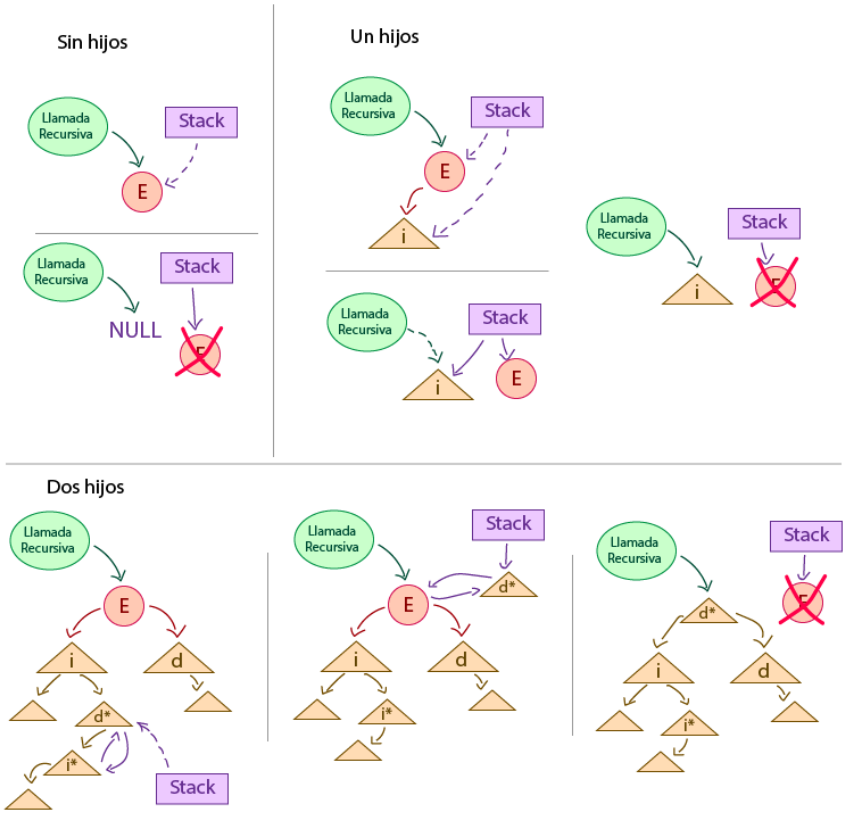

Figura 2: Casos de eliminación.

- **Recorrer:** Realizar cierta acción con todos los elementos del árbol. Existen varias formas de hacer esto, pero por lo general se usan 3:
  - *Preorder:* se recorre el elemento actual, luego todos los que están a la izquierda (siguiendo el mismo orden) y por último los que están a la derecha. Con este recorrido se puede crear una copia del árbol con los mismos elementos y la misma estructura.
  - *Inorder:* se recorre primero el subárbol de la izquierda, luego el elemento actual, y luego el subárbol de la derecha. Este recorrido pasa por los elementos en orden de menor a mayor.

- *Postorder:* se recorren los elementos a la izquierda, luego los de la derecha, y por último el elemento actual. Si se van eliminando todos los elementos del árbol en este orden, siempre se eliminan elementos sin hijos, que es la forma mas eficiente de destruir el árbol.

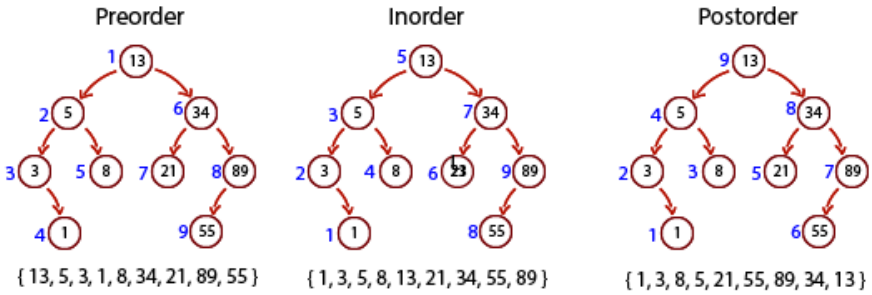

Figura 3: Recorridos de un abb.

Estas son las bases de la implementación más sencilla de un árbol binario de búsqueda. Existen otros tipos de árbol parecidos que son más eficientes en memoria o que resuelven algunos problemas de los abb's como el desbalanceo, pero esta sigue siendo suficiente para muchas aplicaciones y por lo tanto también es muy usada.

3. Detalles de implementación

Como ya se adelantó antes, debido a la naturaleza recursiva de los abb, la mayoría de las funciones fueron implementadas de esa forma. Los nodos del árbol guardan 3 punteros: uno a cada hijo y uno al elemento que almacenan. La estructura del árbol en sí guarda la referencia al nodo raíz, la función comparadora que usa para ordenar, y el tamaño. A continuación se presenta una breve explicación de las funciones implementadas:

- **Crear:** Recibe un comparador y crea un árbol nuevo, con raíz vacía, tamaño 0 y el comparador pasado por el usuario. Devuelve una referencia a ese árbol.
- **Insertar:** Esta función llama a una función recursiva auxiliar que recibe un nodo (la raíz, en el caso del primer llamado), el comparador y el elemento a insertar. La condición de corte es que el nodo con el que se llame sea nulo, en cuyo caso inserta el elemento creando un nuevo nodo con el elemento a insertar y devolviendo eso. Si no se cumple esa condición, vuelve a llamar a la función pero con el hijo izquierdo o el derecho, dependiendo de si el comparador dice que el elemento a insertar es mayor o menor que el actual. En este caso, se decidió que los elementos duplicados se inserten a la izquierda. La función recursiva en sí devuelve nodos del árbol, lo que permite alterar en cierto sentido no sólo el nodo sobre el que se está operando, sino que también (un hijo de) el nodo que realizó la llamada (el padre) por medio del return.
- **Quitar:** Posiblemente la función más complicada. La parte recursiva funciona de manera parecida a la función insertar, solo que se le suma la condición de corte de encontrar el elemento a eliminar. Si la función corta porque llegó a un elemento nulo significa que no existe el elemento a eliminar y no tiene que hacer nada. Pero si encuentra el elemento, pasa al proceso de borrado, que consiste en lo siguiente:

  1. Chequea si el nodo a eliminar tiene un hijo izquierdo. Si no tiene, elimina el nodo y devuelve el hijo derecho (que puede o no existir, pero es indistinto).

2. Chequea si el nodo a eliminar tiene un hijo derecho. Si no tiene, elimina el nodo y devuelve el hijo izquierdo.
3. En este punto, ya sabemos que el nodo a eliminar tiene dos hijos, se procede entonces a buscar el predecesor inmediato. Esto se hace con otra función recursiva auxiliar que recibe el hijo izquierdo y recorre siempre por la derecha hasta encontrar un elemento sin hijo derecho. Ese elemento lo guarda en una variable pasada por referencia y devuelve el hijo izquierdo (que puede o no existir, es indistinto).
4. Se asignan los del nodo a eliminar al predecesor, se elimina el nodo y se devuelve el predecesor.

Como los elementos iguales se consideran menores a la hora de insertar, utilizar el predecesor tiene la ventaja de que si se elimina un elemento duplicado del árbol es reemplazado por otro igual. La forma en la que la función original sabe si se eliminó un elemento o no para actualizar el tamaño de ser necesario es con un booleano que se va pasando como referencia.

- **Buscar:** Es prácticamente idéntica a quitar, solo que cuando encuentra el elemento no lo elimina sino que simplemente lo devuelve.
- **Destruir:** También es recursiva. Si se llama con un nodo nulo corta, si no primero manda a destruir los hijos (por lo que ya fue explicado en la sección teórica), y luego aplica la función destructora sobre el elemento de ser necesario y libera la memoria del nodo. Por último, la función principal (la que llama directamente el usuario y llama a la recursiva) se encarga de liberar la memoria del árbol en sí.
- **Recorrer:** Existen dos funciones de recorrido, una que aplica una función a cada elemento hasta que dicha función devuelva false o recorra todos los elementos, y otra que añade los elementos recorridos a una array hasta que se llene o añada todos los elementos. Ambas funcionan de manera parecida. También reciben el recorrido a seguir, inorder, preorder o postorder, que lo único que afectan es en el orden en el que se hacen los llamados recursivos y el recorrido del elemento. La versión que aplica una función la llama y se guarda el valor que devuelve para saber si seguir recorriendo o no, y la que agrega elementos al array agrega el elemento y actualiza la capacidad. Ambas también actualizan un contador de elementos recorridos que son los que devuelven las funciones principales.
- **Tamanio/Vacío:** Funciones sencillas que simplemente devuelven lo que dice su nombre.