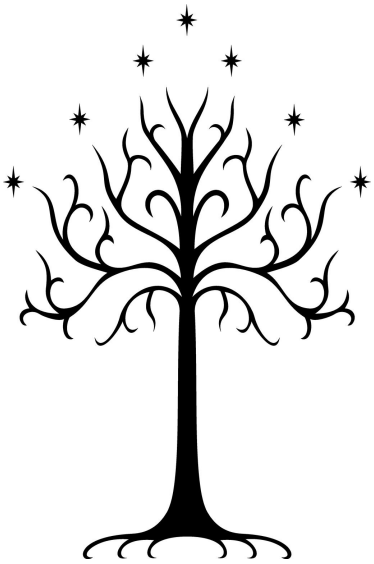


TDA Árbol Binario de Búsqueda

[7541/9515] Algoritmos y Programación II

Primer cuatrimestre de 2022

Autor:	**Email**:	**Padrón**:	**Fecha**:
Luca Lazcano	llazcano@fi.uba.ar	107044	22/05/2022

Índice

1. Introducción

Continuando con las implementaciones de los principales TDA, en esta ocasión se tuvo que implementar un Árbol Binario de Búsqueda (ABB), con las especificaciones brindadas por la cátedra.

Además, se acompaña la implementación con este documento, donde se explora teoría pertinent al TDA tratado. Se explicará qué se entiende por árbol, árbol binario, y árbol binario de búsqueda; su funcionamiento, sus operaciones y las distinciones entre ellos.

Asimismo, se explicará la implementación realizada y decisiones de diseño tomadas.

Fueron provisto el archivo `abb.h` con las firmas y descripciones de las funciones a desarrollar, y un archivo `pruebas.c` donde se deben implementar pruebas para verificar el funcionamiento esperado de la implementación.

2. Árboles

Un árbol es una estructura de datos que consiste en una colección de nodos. Cuando no está vacío, consiste en un nodo que se distingue del resto, llamado raíz, al cual pueden estar conectados otros nodos. Cada uno de estos se considera el nodo raíz de un subárbol. A cada nodo conectado a una raíz se le llama hijo, y al nodo raíz de estos, nodo padre. Se llama nodos hermanos a nodos con el mismo padre.

Como vemos, esta definición ilustra la naturaleza recursiva de esta estructura, donde cada nodo se considera la raíz de un subárbol distinto.

Un árbol se define entonces por su raíz.

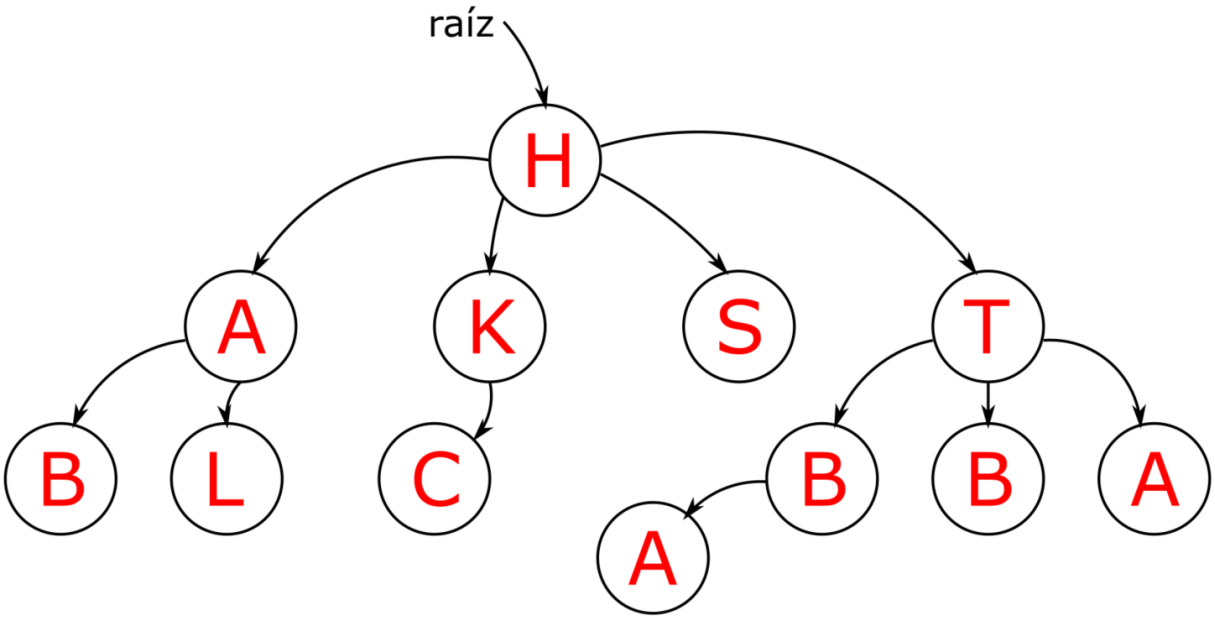


Figura 1: Árbol.

2.1. Árboles binarios

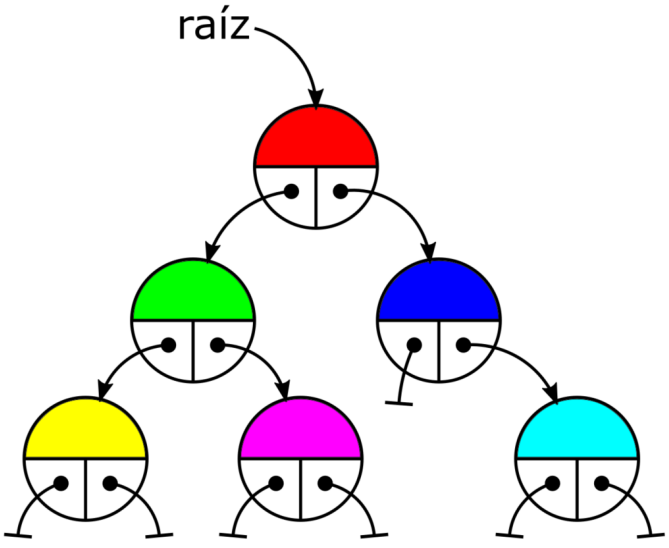


Figura 2: Árbol binario

Un árbol es binario cuando sus nodos pueden tener a lo sumo dos hijos. Los nodos hijos se llaman nodo izquierdo y nodo derecho.

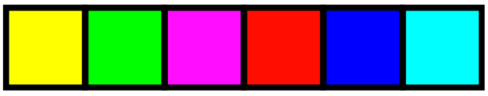
2.1.1. Operaciones

Las operaciones básicas de un árbol binario son: crearlo, destruirlo, recorrerlo y verificar si está vacío; y también se debe poder insertar, eliminar y buscar un elemento.

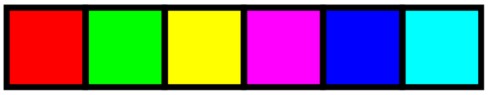
2.1.2. Recorridos

Recorrer un árbol consiste en visitar todos los elementos del árbol de cierta forma. Entre los distintos posibles recorridos se destacan los que llamamos preorden, inorden y postorden.

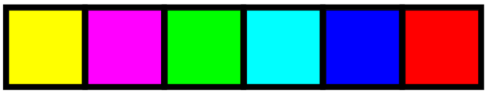
- Inorden: se visita el subárbol izquierdo, luego la raíz, luego el subárbol derecho.



- Preorden: se visita la raíz, luego los subárboles izquierdo y derecho.



- Postorden: se visitan los subárboles izquierdo y derecho, y luego la raíz.



2.2. Árboles binarios de búsqueda

Un árbol binario de búsqueda (ABB), es un árbol binario con la característica extra de que los elementos se ordenan tal que para cada nodo, el elemento a su izquierda (si

existe) es menor, y el de su derecha (si existe) es mayor. Además, cada subárbol es a su vez un ABB.

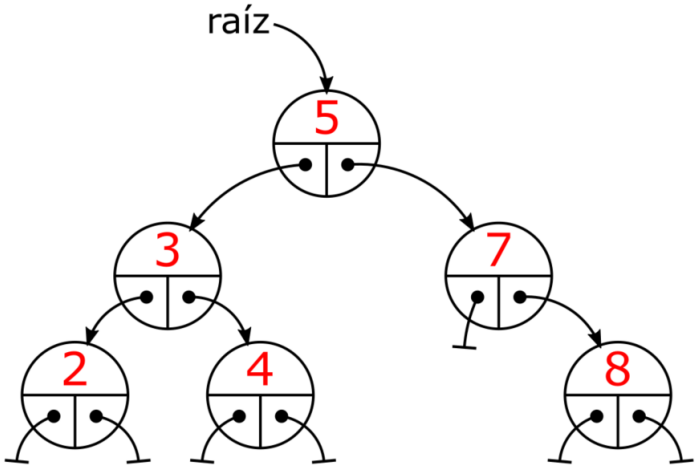


Figura 3: Árbol binario.

Los ABB tienen la ventaja de la velocidad de un árbol binario ($\log(n)$), y de las listas enlazadas en cuanto a su versatilidad para la inserción y extracción de elementos.

Las operaciones posibles son las mismas que en los árboles binarios.

Los recorridos más importantes son los mismos que un árbol binario, pero considerando la característica de orden, los recorridos tienen usos muy particulares.

- Inorden: con este recorrido visitamos los elementos en orden creciente, por lo que es útil para obtener los elementos ordenados. Resultado de recorrer el árbol de la figura 3 y guardar sus elementos en un vector: `[ 2, 3, 4, 5, 7, 8 ]`
- Preorden: este recorrido es útil si buscamos un elemento en particular, ya que verificamos si el elemento en cuestión se encuentra en el nodo antes de seguir adentrándose en el árbol. `[ 5, 3, 2, 4, 7, 8 ]`
- Postorden: este recorrido es útil si queremos eliminar los elementos del árbol, ya que destruimos los subárboles y por último la raíz. `[ 2, 4, 3, 8, 7, 5 ]`

2.3. Funcionamiento de las operaciones

2.3.1. Crear

Se crea un árbol, alocando espacio para un puntero a la raíz, y de ser necesarios una función comparadora, destructora y un contador de elementos.

Complejidad: O(1), ya que se trata de una operación única.

2.3.2. Buscar

1. Se compara el elemento con el elemento en la raíz.
2. Si son iguales, la búsqueda finalizó.
3. Si el elemento buscado es menor, se vuelve al primer paso con el subárbol izquierdo. Si es mayor, con el subárbol derecho.

Complejidad: O(log(n)), ya que en cada comparación se excluyen la mitad de los nodos donde podría encontrarse el elemento buscado. Esta complejidad asume que el árbol se encuentra más o menos balanceado. Cuanto más se degenera el árbol en lista, más tenderá la complejidad a O(n).

2.3.3. Insertar

Consiste en agregar un elemento al árbol.

1. Comparamos el elemento con el elemento en la raíz, si es menor avanzar al subárbol izquierdo, si es mayor, al subárbol derecho.
2. Repetir el paso anterior hasta llegar al final del subárbol donde se ubica el elemento enlazándose desde la raíz donde fue llamado.
3. Se reserva memoria para un nuevo nodo, con el elemento a insertar, y se apuntan sus punteros derecha e izquiera a nulo.

4. Se coloca este nodo como hijo derecho o izquierdo según corresponda del nodo en el cual se estaba posicionado.

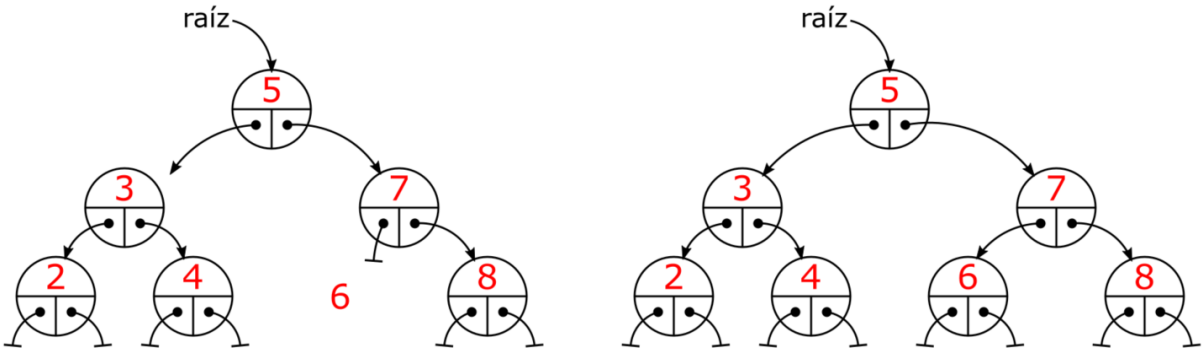


Figura 4: Insertar.

Complejidad: O(log(n)), ya que en cada comparación se excluyen la mitad de los subárboles en donde podría insertarse el elemento. Esta complejidad asume que el árbol se encuentra más o menos balanceado. Cuanto más se degenera el árbol en lista, más tenderá la complejidad a O(n).

2.3.4. Quitar

Consiste en quitar un elemento al árbol. Se puede considerar una extensión de la operación de búsqueda. Una vez encontrado el elemento se deben considerar las siguiente posibilidades:

- Caso 1: el nodo no tiene hijos: es hoja. Entonces se elimina, asignando a nulo el puntero que lo apuntaba.

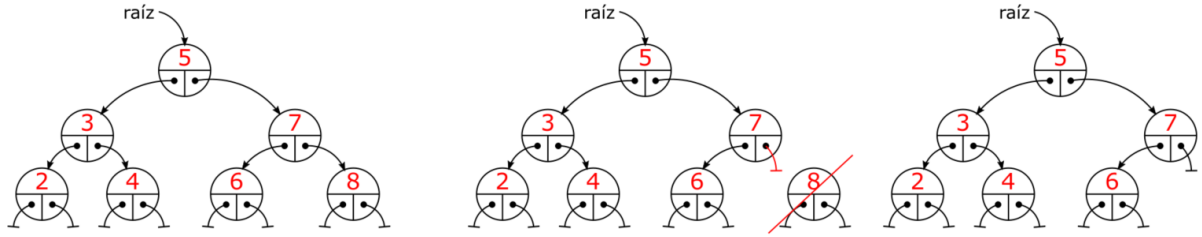


Figura 5: Quitando el 8.

- Caso 2: que tenga un sólo hijo. Entonces se apunta el puntero que lo apunta al puntero hijo del nodo a eliminar.

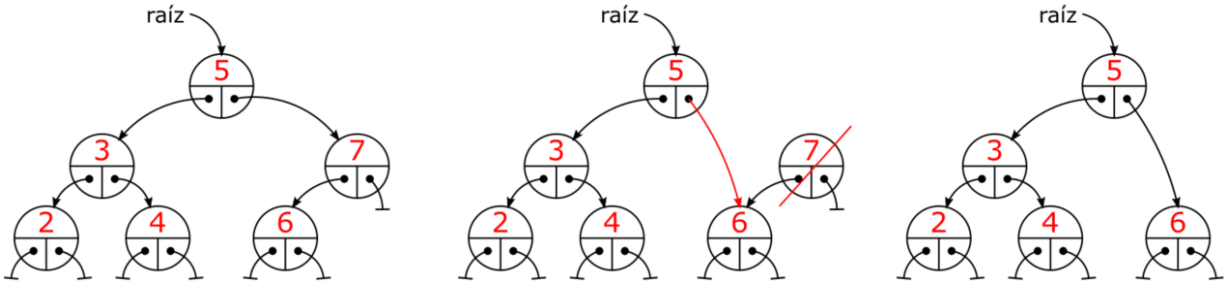


Figura 6: Quitando el 7.

- Caso 3: que tenga dos hijos. Se sustituye el elemento por el predecesor inorden (el más grande del subárbol izquierdo). Se quita este elemento, que será necesariamente caso 1 o 2.

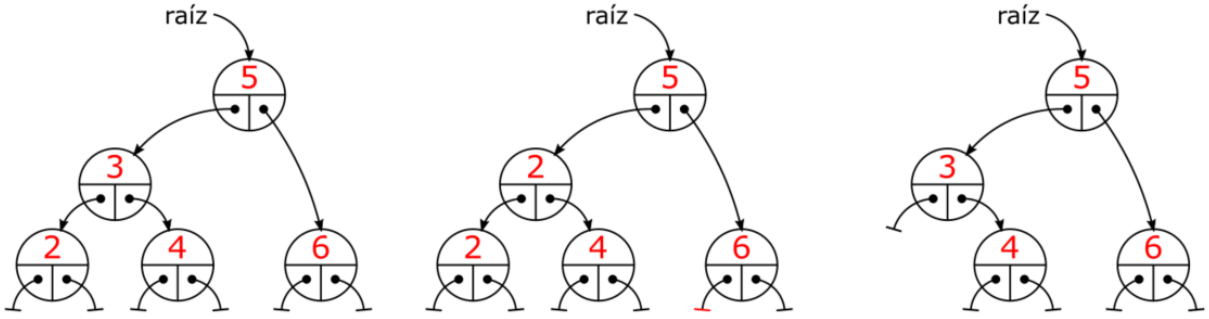


Figura 7: Quitando el 3.

Complejidad: O(log(n)), ya que en cada comparación se excluyen la mitad de los nodos donde podría encontrarse el elemento buscado. Esta complejidad asume que el árbol se encuentra más o menos balanceado. Cuanto más se degenera el árbol en lista, más tenderá la complejidad a O(n).

2.3.5. Recorrer

Para recorrer el árbol, se:

- Visita el elemento actual. Si es una hoja finaliza el recorrido por esta rama. Sino, se continúa recorriendo con los subarboles.
- Recorre con el subárbol izquierdo.
- Recorre con el subárbol derecho.

Estos tres pasos se realizan en el orden precisado por el tipo de recorrido, ya descrito.

Complejidad: O(n), ya que se visitan todos los elementos del árbol.

2.3.6. Destruir

Para destruir el árbol se lo recorre destruyendo el contenido de cada nodo visitado. Por último se destruye la definición del árbol.

Complejidad: O(n), ya que se visitan todos los elementos del árbol.

3. Implementación y decisiones de diseño

En esta ocasión desarrollé funciones auxiliares, varias más que en el TDA lista. Esto se debe a que hice mucho uso de la característica recursiva que tiene este tipo de estructuras, pero para ello necesitaba hacer recursividad con la raíz del árbol y subárboles,

cuando la mayoría de las funciones provistas son de tipo árbol. Esto tiene sentido ya que el usuario necesita el árbol.

Entonces muchas funciones simplemente realizan una validación rápida e inmediatamente hacen uso de una función auxiliar similar pero que utilizan la raíz del árbol. Por último, devuelven el árbol actualizado.

Las principales funciones abstraídas a funciones auxiliares son:

3.1. `abb_insertar()` y `nodo_abb_insertar()`

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

Esta función recibe un árbol y un elemento a insertar, y devuelve el árbol con el elemento insertado en caso de éxito, o `NULL` en su defecto, fue utilizada para actualizar el nodo raíz llamando a:

```
nodo_abb_t *nodo_abb_insertar(nodo_abb_t *raiz, void *elemento,
↪  abb_comparador, size_t*tamanio)
```

Esta función es recursiva. Se realiza la comparación con la función comparadora pasada por parámetro, y si el elemento a insertar es menor al elemento en la raíz, se actualiza su puntero izquierda llamando recursivamente a la función con el puntero izquierda. Si es mayor, el puntero derecho.

El caso base se da cuando se pasa una raíz nula, es decir, se ha llegado al fin del árbol y esta raíz apuntará a la nueva hoja. Entonces se crea un nuevo nodo iniciando sus campos en `NULL`, se asigna su puntero elemento al elemento a insertar, se aumenta el contador del tamaño pasado por parámetro y se devuelve la nueva hoja.

Por la forma en que fue recursivamente llamada la función, se habrá actualizado el árbol y se devolverá su raíz.

3.2. `abb_quitar()`, `nodo_abb_quitar()` y `nodo_max`

```
void *quitar(abb_t *arbol, void *elemento)
```

Esta función recibe un árbol y un elemento a quitar, y devuelve el elemento quitado si lo encuentra o `NULL` en su defecto. Para almacenar el elemente quitado crea un puntero inicializado en `NULL` que pasa como parámetro a la función auxiliar, que actualiza la raíz del árbol:

```
nodo_abb_t *nodo_abb_quitar(nodo_abb_t *raiz, void *elemento,
↪  abb_comparador, size_t*tamanio, void **quitado)
```

Esta función es recursiva. Realiza la comparación entre el elemento en la raíz y el elemento a quitar. Si el elemento buscado es menor o mayor se procede recursivamente como en insertar y buscar.

Si la comparación da 0, es que se ha encontrado el elemento a quitar. Aquí pueden darse alguno de los tres casos ya descritos en la teoría. Si el elemento es una hoja o un nodo con un sólo hijo, se apunta al elemento con el puntero `quitado`, y se actualiza la raíz a `NULL` o el hijo, según corresponda.

Si se da el caso 3, se apunta `quitado` al elemento, y luego este al predecesor inorden (para ello se realizó otra función auxiliar, `nodo_max()`). Luego se actualiza la raíz izquierda llamando recursivamente pero buscando eliminar el elemento predecesor inorden, con los punteros `tamanio` y `quitado` como `NULL`, ya que no deben ser modificados en este llamado. Esta solución reduce el caso 3 al caso 1 o 2.

El árbol se encuentra actualizado y se devuelve su raíz.

3.3. `abb_buscar()` y `nodo_abb_buscar`

Este par de funciones sigue la misma lógica que las ya descritas. `abb_buscar` valida que árbol no sea `NULL`, y devuelve el elemento encontrado por `nodo_abb_buscar()` que

recibe la raíz, e itera hasta encontrar el elemento, que devuelve.

3.4. `abb_destruir()`, `abb_destruir_todo()` y `nodo_abb_destruir_todo()`

Siguiendo la lógica previa, `abb_destruir_todo()` usa `nodo_abb_destuir_todo()` que iterativamente destruye todos los elementos y libera la memoria asignada a cada nodo.

Para `abb_destruir()` se utilizó `abb_destruir_todo()` pasando como función destructora `NULL`.

3.5. Funciones iteradoras

Las funciones iteradoras (`abb_recorrer()` y `abb_con_cada_elemento()`) constan básicamente de un `switch case` que llama a funciones auxiliares que realizan el tipo de recorrido solicitado.

En el caso de `abb_con_cada_elemento()`, sus funciones auxiliares son de tipo `bool` para validar si continuar el recorrido o detenerse.