FACULTAD DE INGENIERIA

Universidad de Buenos Aires

TDA — ABB

[7541/9515] Algoritmos y Programación II

Segundo cuatrimestre de 2021

Alumno:	Hurtado, Daniel
Número de padrón:	107404
Email:	dhurtado@fi.uba.ar

1. Introducción

Este trabajo práctico tiene como objetivo la continuación en las implementaciones de TDA (Tipo de dato abstracto), proceso iniciado en el TDA Lista anterior. Recordando la anterior explicación, podemos retomar la definición de un TDA como la creación de estructuras y funcionalidades definidas que no hagan referencia a datos específicos o contenido irrelevante para su propósito (abstracción), brindando la posibilidad de ser reutilizados en infinitud de situaciones que requieran dichas funcionalidades, sin importar la índole de la información, datos o procesos manejados en donde se requiera implementarla.

En este TDA, se implementará las definiciones de ABB (Arbol Binario de Búsqueda), siguiendo de igual manera el desarrollo enfocado en TDD (Test Development Drive, o en español, desarrollo guiado por pruebas), cuya finalidad es realizar la implementación conociendo de antemano cual sería su correcto y esperado funcionando, a la vez que dicha implementación seguirá los criterios establecidos por la cátedra y el contrato facilitado (tda_abb.pdf y abb.h).

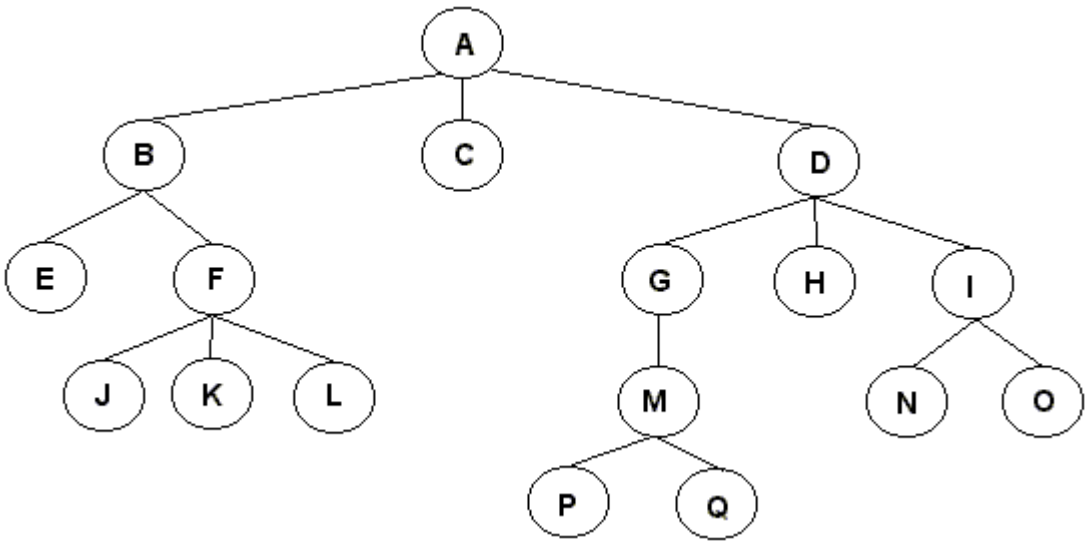
2. Teoría

Previo a profundizar en el ABB a implementar, podemos permitirnos un recordario del por qué de su existencia, en general de los TDA. En un lenguaje como C, cuyos tipos de datos y funcionalidades son limitadas con respecto a otros lenguajes más completos en estos aspectos como Python o Javascript; los programadores carecemos de opciones nativas de utilizar, o en este caso, organizar de mejor manera gran cantidad de información de manera accesible y ordenada, sin depender a su vez de las limitaciones de memoria de estos lenguajes fuertemente tipados. Debido a esta razón, se realizan implementaciones de nuevos tipos de datos que nos permitan organizar, agrupar y manejar datos de diferentes tipos.

1. Árboles en general

Los árboles son tipos de datos de almacenamiento de información diversa, las cuales se encuentran organizadas entre si de manera jerárquica, es decir, la organización de las mismas deja de ser lineal (una seguida de otra) como sucede en las listas, pilas y colas, a pasar a una organización flexible, con datos que pertenezcan (o desciendan) de otros datos de manera ilimitada (salvo limitaciones de hardware y cómputo).

Tienen la forma de árboles cuyos datos "hijos" decienden de ellos a través de conexiones entre nodos de memoria similar a ramas entre ellos, de la siguiente manera:



En una estructura de árbol podemos definir los siguientes términos:

- Nodo Raíz o simplemente raíz: Es aquel nodo o dato con el que se da inicio al árbol, siendo el punto más alto en la jerarquía de la organización.
- Nodo Hijo: Son aquellos nodos o datos que no son raíces, es decir, tienen obligatoriamente (ya que de lo contrario no se podría llegar a ellos) nodos a un nivel por encima (padres).
- Nodo Padre: Contrario al nodo hijo, son los nodos que tienen hijos y por tanto, tienen información decendiendo de ellos.

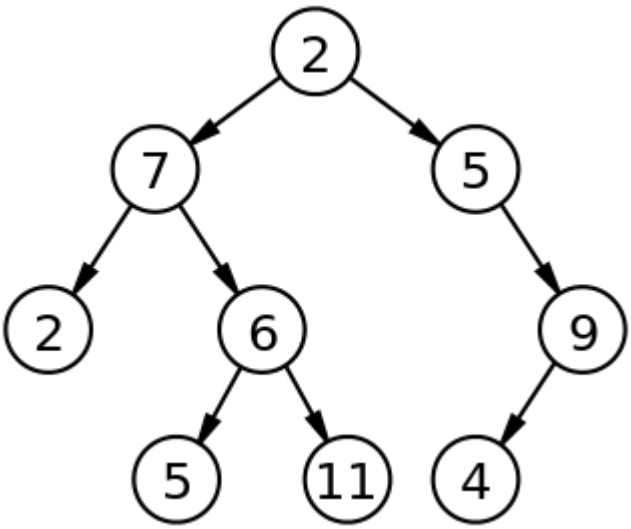
- Nodo Hoja: Son aquellos nodos que no tienen hijos o descendencia.

Además, existen ciertas restricciones en la definición de un arbol:

- Un nodo no puede tener más de un padre.
- La información debe ser descenciente desde la raiz a los nodos hojas. No puede existir bucles ni conexiones "circulares" entre los nodos.

2. Árboles binarios

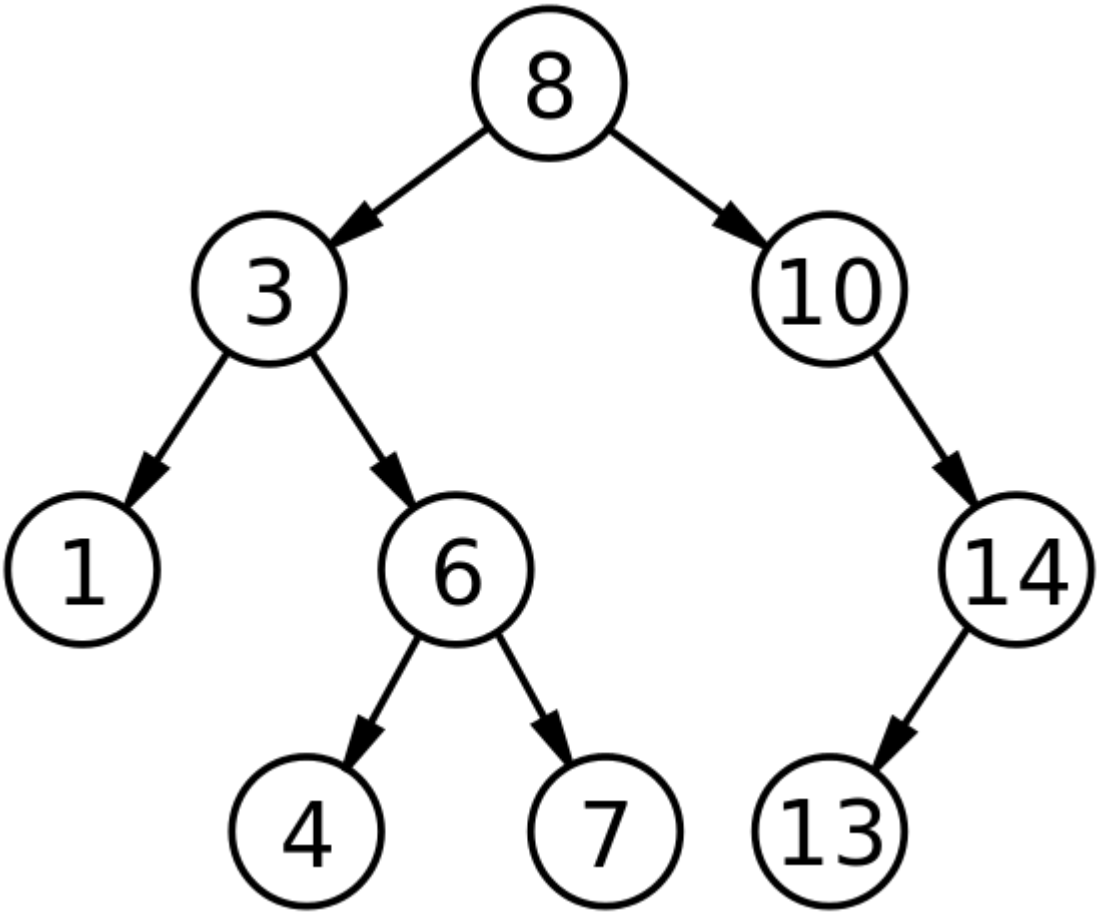
Los árboles binarios son un tipo de implementación de árboles en los cuales cada nodo solo es capaz de tener un máximo de 2 nodos hijos (pudiendo no tener, o tener uno solo). Son utilizados con el objetivo de establecer una estructura "estricta" del árbol, en lugar de brindar la posibilidad de infinitud de hijos.



3. Árboles binarios de búsqueda (ABB)

Estos tipos de árboles binarios son aún más específicos, como su nombre indica, están enfocados en ser eficientes para la búsqueda de información gracias a mantener una estructura interna clara y consistente en cuanto al posicionamiento relativo de los datos contenidos dentro de ellos.

La organización de los mismos viene dada por la dirección de los hijos de cada nodo, al tener dos, pueden ser difinidos como hijo izquierdo e hijo derecho. Comunmente son utilizados siguiente el criterio de posicionamiento de elementos menores (que el nodo actual evaluado), a la izquierda del mismo, y los mayores a la derecha.



Para una correcta definición abstracta de un ABB que pueda ser utilizado independientemente de los elementos contenidos, su creación requiere de una función de comparación definida, con el objetivo de organizar los datos relativamente entre sí en base a los criterios necesarios o establecidos.

En cuanto a la complejidad, los ABB permiten optimizar (con respecto a datos no jerárquicos como la lista, pila y cola) el procesamiento de ciertas acciones como búsqueda e inserción de datos, pasando de una complejidad de O(n) a O(log n) gracias a conocer donde debería estar ubicado el elemento buscado/insertado comparando los actuales, diviendo cada vez menos la cantidad de pasos a dar, sin ser necesario recorrer la totalidad del árbol como sucedería con una lista.

3. Detalles de implementación

Antes de entrar a detalle en las diversas funciones y lógica realizada para la implementación solicitada, y debido a que los tipos de datos puede variar entre una implementación y otra, podemos empezar describiendo la base o contrato de nuestro ABB y nodos:

```
typedef enum {INORDEN, PREORDEN, POSTORDEN} abb_recorrido;

/**
* Comparador de elementos. Recibe dos elementos y devuelve 0 en caso de ser
* iguales, >0 si el primer elemento es mayor al segundo o <0 si el primer
* elemento es menor al segundo.
*/
typedef int (*abb_comparador)(void*, void*);
```

```
typedef struct nodo_abb {
  void* elemento;
  struct nodo_abb* izquierda;
  struct nodo_abb* derecha;
} nodo_abb_t;

typedef struct abb{
  nodo_abb_t* nodo_raiz;
  abb_comparador comparador;
  size_t tamanio;
} abb_t;
```

Entre los requerimientos de la implementación más destacados, podemos mencionar:

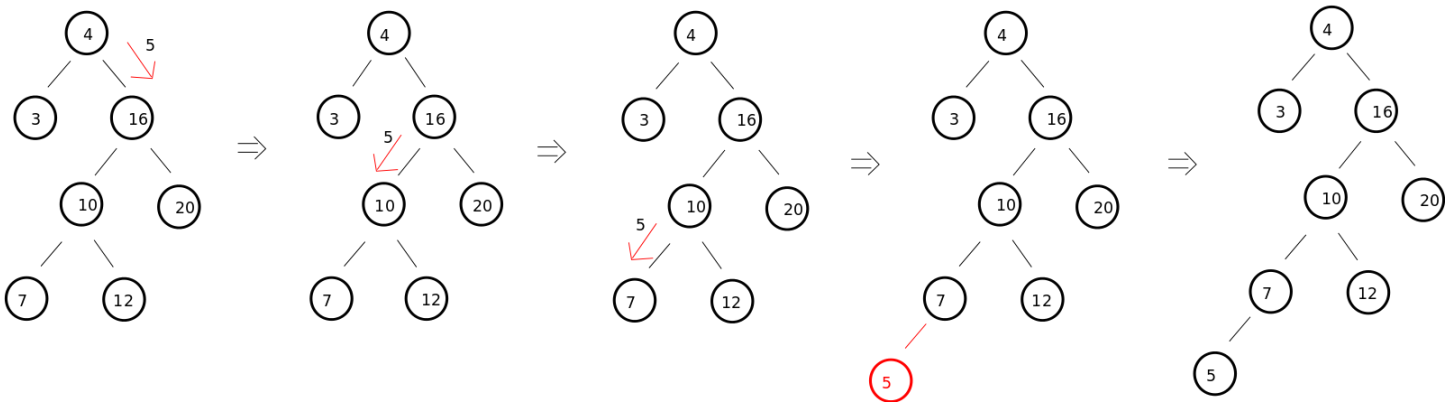
1. Inserción de elementos

Conociendo la estructura de nuestro árbol binario de búsqueda, sabemos que empezamos por el nodo raíz (nuestro abb empieza sin raíz, y el primer elemento insertado será la raiz), y lo recorremos hasta hallar la posición en la que deberá estar nuestro elemento insertado, según el orden y el comparador establecido en la creación del mismo.

Recordando, si es te elemento es menor, vamos a la izquierda del nodo actual, y si es mayor vamos a la derecha del nodo actual. Este es un proceso repetido a lo largo del recorrido entre los nodos hasta llegar al "final", donde será insertado el elemento.

Este recorrido (que también sería la misma lógica para búsqueda y eliminación) es realizado recursivamente, evaluando cada nuevo nodo descendido de la misma manera ya que no se requiere algún cambio adicional.

Un ejemplo del funcionamiento, insertando el número 5 en un árbol con elementos:



Podemos observar como vamos recorriendo el arbol, empezando desde la raíz (4) moviéndonos a la derecha por ser mayor, y luego a la izquierda por ser menor que 16, y consecutivamente hasta llegar a 7, el número mayor más chico que 5, en donde incertamos el 5 y vinculamos al 7 por la izquierda (por ser menor).

2. Eliminación de elementos

Similar a la inserción, requerimos recorrer primero el árbol hasta encontrar la posición (o no hacer nada si no se encuentra) en la que se encuentre para eliminar dicho nodo. En esta implementación, y por requerimiento de la cátedra, el movimiento en los nodos al eliminar uno, consiste en reemplazar dicho nodo por el predecesor inmediato inorder (o más entendiblemente, "el más grande de los menores"), el cual podemos encontrarlo como el último descendiende derecho de la raíz izquierda el elemento a eliminar.

Yendo a la práctica, existen diversos casos a considerar en una eliminación, según sea el nodo que buscamos eliminar:

1- Nodo sin hijos: Es el caso más sencillo, bastará con destruir este nodo y eliminar la referencia a él de su nodo padre.

2- Nodo con un único hijo: Bastará con eliminar al nodo, y reemplazar la referencia que su padre tenía a él, por la referencia del hijo del nodo a eliminar.

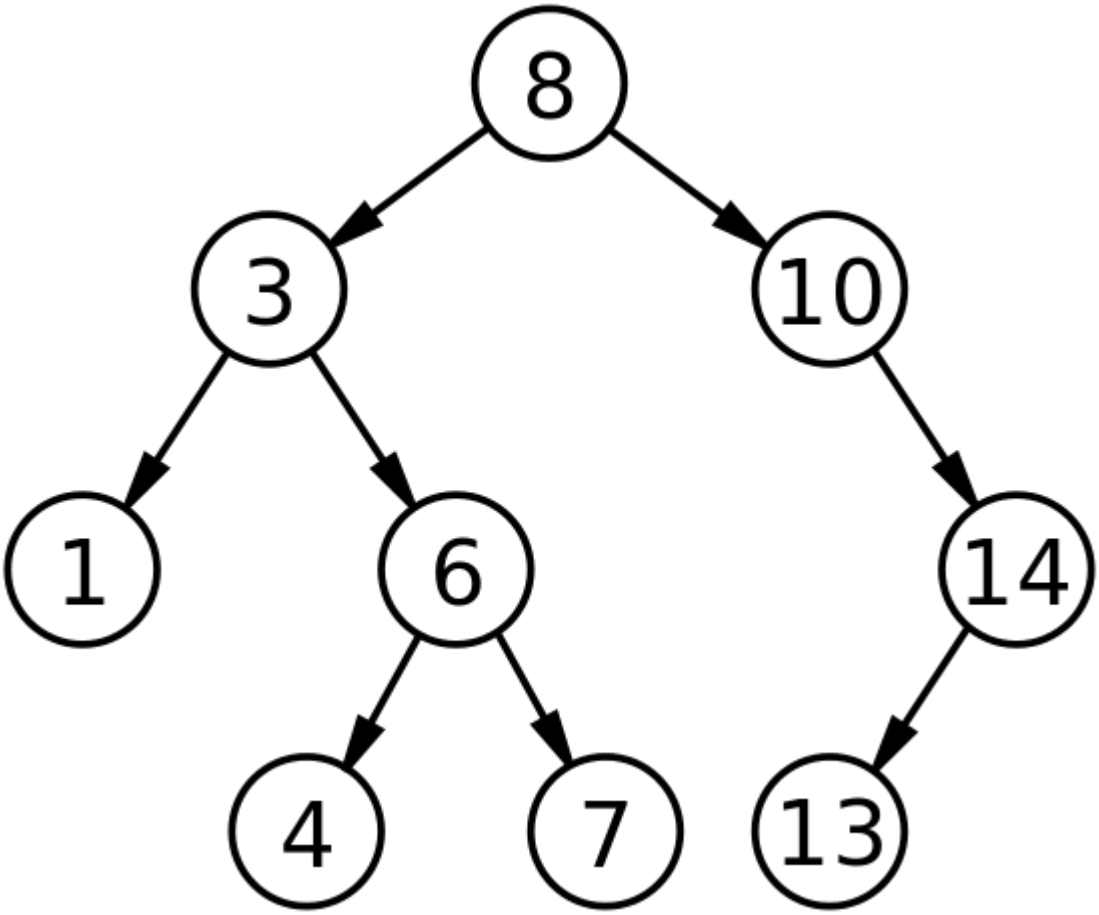
3- Nodo con dos hijos: Es el caso más complejo. Luego de conseguir el predecesor del elemento a eliminar, y sabiendo que este será el que reemplazará el lugar del eliminado, debemos reemplazar la referencia a él de su antiguo padre por su rama izquierda (para no moverla toda hacia arriba complicando el procedimiento), y asignar la referencia del nodo que estamos moviendo por la izquierda a la referencia del nodo que estamos eliminando para no perder el extremo del árbol. Hecho esto, podemos posicionar este nuevo nodo a mover en la posición deseada y actualizar la referencia del padre del nodo eliminado a él.

Adicionalmente, existe la posibilidad de que el nodo a reemplazar sea el nodo raíz, por lo que sumado a los casos anteriores, debemos actualizar la raíz del arbol por el nuevo nodo (bajo el mismo criterio de predecesor inmediato inorder) y asignar la referencia derecha de la anterior raiz a la nueva para no perder dicho extremo del árbol.

3. Iteración/Recorrido de elementos:

Nuestra implementación brinda la posibilidad de iterar o recorrer nuestro elemento mediante dos implementaciones, la primera haciendo uso de una función definida por el usuario cuyo recorrido se detendrá cuando dicha función devuelva false (abb_con_cada_elemento), y también mediante una función que recorra y almacene los elementos del recorrido dentro de una lista (abb_recorrer).

Para ello se implementaron 3 criterios de recorrido, ejemplificados con la siguiente imagen:



1- Inorder: Se visita el último subarbol izquierdo, seguidamente del último subárbol derecho seguido del nodo por el que se ingresó y posteriormente el subárbol derecho. En la imagen anterior, este recorrido estaría descrito como: 1, 3, 4, 6, 7, 8, 10, 13, 14.

2- Preorder: Se visita primero el nodo actual, seguido del último subárbol izquierdo y terminando con el subárbol derecho. En la imagen anterior este recorrido estaría descrito como: 8, 3, 1, 6, 4, 7, 10, 14, 13.

3- Postorder: Se visita el subárbol izquierdo, seguido del subárbol derecho y termina con el nodo por el que se ingresó. En la imagen anterior este recorrido estaría descrito como: 1, 4, 7, 6, 3, 13, 14, 10.

4. Utilización y pruebas

Para utilizar este tda, necesitamos importar la librería abb.h en nuestro programa, agregando ambos archivos (abb.c y abb.h) a nuestro script de compilación. Una vez realizado, podemos empezar a crear un árbol, no antes de contar con una función comparadora para la información que planeamos guardar.

Por ejemplo, si nos interesa almacenar números enteros, podemos utilizar una función comparadora similar a esta:

```
int comparador_int(void* num1, void* num2) {
  int valor1 = *((int*) num1);
  int valor2 = *((int*) num2);
```

```
  if (valor1 == valor2) return 0;
  return (valor1 < valor2) ? -1 : 1;
}
```

Y posteriormente crear nuestro arbol mediante:

```
abb_t* arbol = abb_crear(&comparador_int);
```

Hecho esto, podemos empezar a utilizarlo como almacenamiento de información, agregando y borrando respectivamente haciendo uso de las siguientes funciones, descritas además en abb.h:

```
/**
* Inserta un elemento en el arbol.
* El arbol admite elementos con valores repetidos.
*
* Devuelve el arbol en caso de exito o NULL en caso de error.
*/
abb_t* abb_insertar(abb_t* arbol, void* elemento);

/**
* Busca en el arbol un elemento igual al provisto (utilizando la funcion de
* comparación) y si lo encuentra lo quita del arbol y lo devuelve.
*
* Devuelve el elemento extraido del árbol o NULL en caso de erro.
*/
void* abb_quitar(abb_t* arbol, void* elemento);
```

Podemos iterar mendiante:

```
/**
* Recorre el arbol e invoca la funcion con cada elemento almacenado en el mismo
* como primer parámetro. El puntero aux se pasa como segundo parámetro a la
* función. Si la función devuelve false, se finaliza el recorrido aun si quedan
* elementos por recorrer. Si devuelve true se sigue recorriendo mientras queden
* elementos.
*
* Recorrido especifica el tipo de recorrido a realizar.
*
* Devuelve la cantidad de veces que fue invocada la función.
*/
size_t abb_con_cada_elemento(abb_t* arbol, abb_recorrido recorrido, bool (*funcion)(vo

/**
* Recorre el arbol según el recorrido especificado y va almacenando los
* elementos en el array hasta completar el recorrido o quedarse sin espacio en
* el array.
*
* El array tiene un tamaño maximo especificado por tamanio_array.
*
```

```
 * Devuelve la cantidad de elementos que fueron almacenados exitosamente en el
 * array.
 */
size_t abb_recorrer(abb_t* arbol, abb_recorrido recorrido, void** array, size_t tamani
```

Y finalizar, destruyendo con:

```
/**
 * Destruye el arbol liberando la memoria reservada por el mismo.
 */
void abb_destruir(abb_t* arbol);

/**
 * Destruye el arbol liberando la memoria reservada por el mismo.
 *
 * Adicionalmente invoca el destructor en cada uno de los elementos almacenados
 * en el arbol (si la funcion destructor no es NULL).
 */
void abb_destruir_todo(abb_t* arbol, void (*destructor)(void*));
```