**Árboles AVL**

Kevin Mauricio Carmona

**Estructuras de datos**

**Universidad tecnológica de Pereira**

**Noviembre 15**

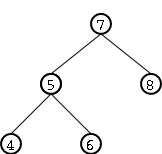
**Año 2019**

**Árboles AVL:**

**Definición:** Un árbol AVL, Es un árbol binario de búsqueda que cumple con la condición de que la diferencia entre las alturas de los subárboles de cada uno de sus nodos es, como mucho 1.

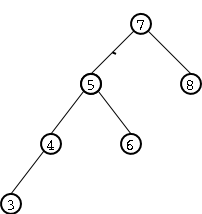
La denominación de árbol AVL viene dada por los creadores de dicha estructura, (Adelson-Velskii y Landis).

Recordamos que un árbol binario de búsqueda es un árbol binario en el cual cada nodo cumple con que todos los nodos de su subárbol izquierdo son menores que la raíz y todos los nodos del subárbol derecho son mayores que la raíz.



**Fig 1.** Árbol AVL de enteros.

A modo de ejemplificar esta dificultad, supongamos que al árbol AVL de enteros de Fig 1 le queremos agregar el entero 3. Si lo hacemos con el procedimiento normal de inserción de árboles binarios de búsqueda el resultado sería el árbol de Fig 2 el cual ya no cumple con la condición de equilibrio de los árboles AVL dado que la altura del subárbol izquierdo es 3 y la del subárbol derecho es 1.



**Fig 2.** Árbol que no cumple con la condición de equilibrio de los árboles AVL.

**Declaración de tipo de dato:**

Iremos ya declarando el tipo de dato que representará un árbol AVL. Esto nos ayudará a formalizar las cosas y nos permitirá en el correr de este documento ir definiendo las operaciones sobre el tipo de dato abstracto.

**Consideraciones:**

1. Como se podrá ver en el siguiente listado, la única diferencia de un árbol AVL, con un árbol binario común es la variable altura en la estructura nodo.
2. Los nodos de un árbol pueden almacenar cualquier tipo de dato, arbitrariamente complejo.

A continuación se lista la declaración del tipo abstracto de dato árbol AVL:

struct Nodo

{

int dato;

struct Nodo \*izq;

struct Nodo \*der;

int altura;

} \*arbol==NULL;

Cada nodo tendrá almacenada su propia altura con respecto a la raíz absoluta del árbol con el que estamos trabajando.

A continuación declaramos las operaciones básicas sobre árboles binarios y con las cuales trabajaremos para acceder al tipo abstracto de dato árbol AVL de aquí en más.

/\* Constructores \*/

Nodo \*vacio (void);

/\* devuelve un árbol AVL vacío \*/

Nodo \*hacer (int x, struct Nodo \*izq, struct Nodo \*der);

/\* devuelve un nuevo árbol formado por una raíz con valor x,

subárbol izquierdo el árbol izq y subárbol derecho el árbol

der. \*/

/\* Predicados \*/

bool es\_vacio (struct Nodo \*t);

/\* devuelve true sii. t es un árbol vacío. \*/

/\* Selectores \*/

Nodo \*izquierdo (struct Nodo \* t);

/\* devuelve el subárbol izquierdo de t. \*/

Nodo \*derecho (struct Nodo \* t);

/\* devuelve el subárbol derecho de t. \*/

int raiz (struct Nodo \* t);

/\* devuelve el valor de la raíz del árbol t. Precondición:

!es\_vacio(t) \*/

int altura (struct Nodo \* t);

/\* devuelve la altura del nodo t en el árbol \*/

/\* Destructures \*/

void destruir (struct Nodo \* t, void (\*free\_dato) (int));

/\* libera la memoria ocupada por los nodos del árbol. Si los

datos almacenados en cada nodo están almacenados dinámicamente

y se los quiere liberar también, debe pasarse como segundo

parámetro una función de tipo void func(int t) que libere

la memoria de objetos int. Si los datos no están

almacenados dinámicamente o simplemente no se los quiere

destruir (liberar de memoria), pásese como segundo parámetro

NULL. Nota: Función Recursiva! \*/

**Consideraciones sobre la altura de los nodos:**

Como vimos en la definición del tipo abstracto para nodos de árboles AVL, se necesitará tener acceso a la altura de cada nodo del árbol en tiempo constante. Dado que una función para hallar la altura de un nodo dado de un árbol tendrá un tiempo de ejecución de O(log(n)) peor caso, no nos queda otra alternativa que almacenar una variable altura en cada nodo e irla actualizando en las inserciones y eliminaciones que se efectúen sobre el árbol.

Una función para calcular la altura de un nodo puede escribirse recursivamente como:

int altura(struct Nodo \*t)

{

if(es\_vacio(t))

return -1;

else

return t->altura}

Queremos que la altura de un árbol que consta de sólo un nodo sea 0, Entonces debemos definir la altura de un árbol vacío como -1.

Es muy importante tener cuidado en actualizar el campo altura de cada nodo siempre que modifiquemos de alguna manera el árbol AVL.

Así, es importante tener una función que nos permita actualizar la altura de un nodo cualquiera del árbol y cuyo tiempo de ejecución sea O(1) en el peor de los casos. A continuación se lista una tal función:

void actualizar\_altura (AVLTree \* t)

{

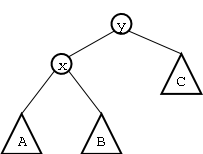
if(!es\_vacio(t))

t->altura = max (altura ((t)->izq), altura ((t)->der)) + 1;

}

**Rotaciones Simples.**

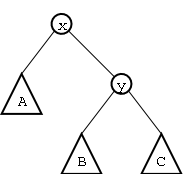
Veremos a continuación una operación sencilla sobre un árbol binario de búsqueda que conserva el orden en sus nodos y que nos ayudará a restaurar la propiedad de equilibrio de un árbol AVL al efectuar operaciones sobre el mismo que pueden perturbarla.



**Figura 3: Árbol antes de la rotación simple.**

Miremos por un momento el árbol de la figura 3. Dado que este es un árbol de búsqueda se debe cumplir que X<Y y además todos los nodos del subárbol A deben ser menores que X y Y; todos los nodos del subárbol B deben ser mayores que X pero menores que Y; y todos los nodos del subárbol C deben ser mayores que Y y por lo tanto que X.

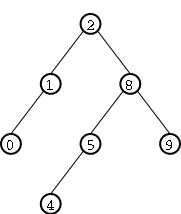
Ahora:



**Figura 4. Árbol luego de rotación simple.**

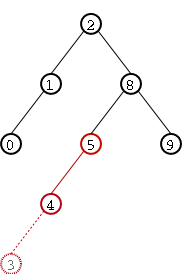
En la figura 4 se ha modificado sencillamente el árbol. Como puede verificarse fácilmente por las desigualdades descriptas en el párrafo anterior, el nuevo árbol sigue manteniendo el orden entre sus nodos, es decir, sigue siendo un árbol binario de búsqueda. A esta transformación se le denomina **rotación simple, (o sencilla).**

Veamos un ejemplo concreto. Deseamos insertar el número 3 en el árbol de enteros:



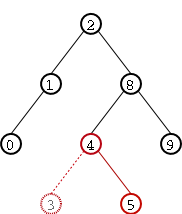
**Figura 5. Árbol AVL.**

La inserción se muestra punteada en color rojo a continuación:



**Figura 6. Árbol luego de la inserción: pérdida de la propiedad de equilibrio marcada con rojo.**

Sin embargo, como puede verse, la inserción ha provocado la pérdida de la propiedad de equilibrio del árbol, ya que dicha propiedad no se cumple en el nodo marcado con rojo, ¿Qué hacemos para recomponer dicha propiedad? Simplemente realizamos una rotación simple. En este caso se dice que la rotación es *izquierda* ya que la “pérdida de equilibrio” se produce hacia la izquierda. A continuación veremos dicha propiedad:



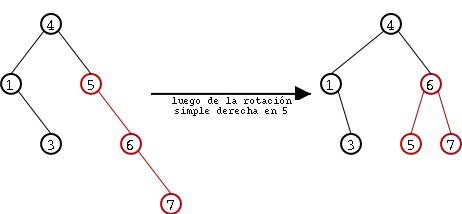
**Figura 7. Restablecimiento de la propiedad de equilibrio mediante una rotación simple sobre el nodo de valor 5.**

En la figura 7, puede verse el árbol luego de la rotación: la propiedad de equilibrio ha sido restablecida. Como mostramos atrás, la rotación conserva el orden entre los nodos, por lo que podemos afirmar que éste último árbol, si es AVL.

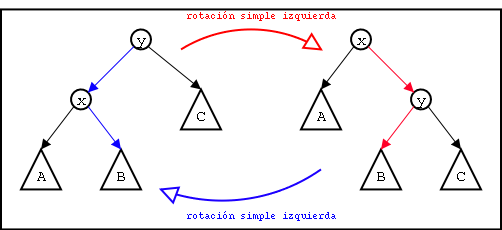
Como podemos observar, el resultado luego de la rotación es un árbol de AVL: posee tanto el orden correcto de un árbol de búsqueda entre sus nodos y la propiedad de equilibrio. En este caso, el “desequilibrio” en el árbol con raíz 5 era enteramente hacia la izquierda y por lo tanto, como ya dijimos, la rotación efectuada se denomina *rotación simple izquierda.* En el caso de un “desequilibro” hacia la derecha, la rotación es análoga y se denomina *rotación simple derecha.*

A continuación se mostrará un pequeño ejemplo por el lado derecho:

**Figura 8. Ejemplo de restablecimiento de propiedad de equilibrio gracias a una rotación simple derecha.**



**Figura 9. Ilustración de la operación de la rotación simple.**



**Implementación de la rotación simple en C:**

void rotar\_s (struct Nodo \*\* t, bool izq);

/\* realiza una rotación simple del árbol t el cual se

pasa por referencia. La rotación será izquierda

sii. (izq==true) o será derecha

sii. (izq==false).

Nota: las alturas de t y sus subárboles serán actualizadas

dentro de esta función!

Precondición:

si (izq==true) ==> !es\_vacio(izquierdo(t))

si (izq==false) ==> !es\_vacio(derecho(t))

\*/

void rotar\_s (Nodo \*\* t, bool izq)

{

struct Nodo \*t1;

if (izq) /\* rotación izquierda \*/

{

t1 = izquierdo (\*t);

(\*t)->izq = derecho (t1);

t1->der = \*t;

}

else /\* rotación derecha \*/

{

t1 = derecho (\*t);

(\*t)->der = izquierdo (t1);

t1->izq = \*t;

}

/\* actualizamos las alturas de ambos nodos modificados \*/

actualizar\_altura (\*t);

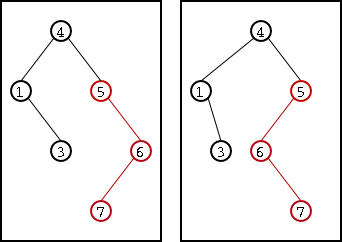
actualizar\_altura (t1);

/\* asignamos nueva raíz \*/

\*t = t1; }

**Rotaciones dobles.**

Hemos visto cómo restaurar la propiedad de equilibrio cuando se presentan desequilibrios “hacia la izquierda” o “hacia la derecha” luego de realizar inserciones en un árbol AVL. Sin embargo y como veremos, pueden ocurrir “desequilibrios en zig-zag”, es decir desequilibrios que no son ni a la derecha ni a la izquierda como es el siguiente caso:

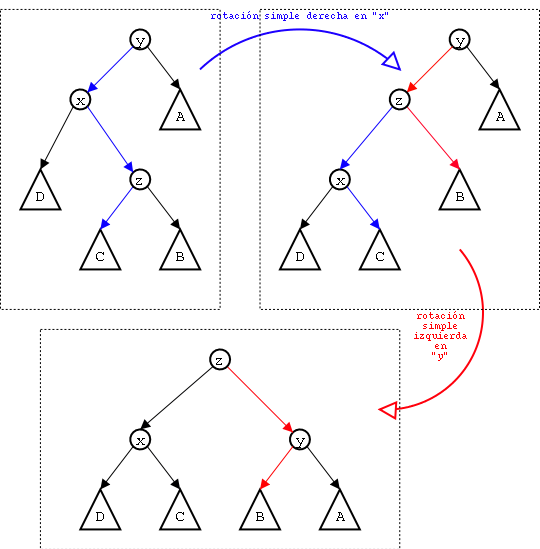


**Figura 10. Ejemplos de desequilibrios en los cuales no funciona rotación simple.**

En estos casos se aplica otro tipo de rotación denominado *rotación doble,* la cual, análogamente a la rotación simple, puede ser izquierda o derecha según sea el caso.

En realidad, la rotación doble constará de dos rotaciones simples. El caso general de la rotación doble izquierda en un árbol AVL se puede observar en la figura 11:

**Figura 11. Rotación doble izquierda.**



La rotación doble derecha es el proceso inverso a la rotación doble izquierda.

void rotar\_d (struct Nodo \*\* t, bool izq);

/\* realiza una rotación doble. Funciona análogamente a

rotar\_s(). \*/

void rotar\_d (struct Nodo \*\* t, bool izq)

{

if (izq) /\* rotación izquierda \*/

{

rotar\_s (&(\*t)->izq, false);

rotar\_s (t, true);

}

else /\* rotación derecha \*/

{

rotar\_s (&(\*t)->der, true);

rotar\_s (t, false);

}

/\* la actualización de las alturas se realiza en las rotaciones

simples \*/

}

Se tiene que la rotación doble, es una rotación simple doblemente recursiva.

**Balance de árbol.**

Como se mostró anteriormente, cada vez que se modifique el árbol (agregar o eliminar elementos) corremos el riesgo de que se pierda su propiedad de equilibrio en alguno de sus nodos, la cual debe conservarse.

La idea general que se utiliza en esta implementación de árboles AVL para implementar los algoritmos de inserción y de eliminación de nodos sobre un AVL, es la siguiente:

* Efectuar los algoritmos de igual forma que en los árboles binarios de búsqueda pero en cada recursión ir actualizando las alturas y re-balanceando el árbol en caso de que sea necesario.

En la sección de consideraciones (ubicada más arriba de este documento), se implementó una función la cual actualiza la altura de un nodo. Así, lo que nos falta es una función que detecte un “desequilibrio” en un nodo dado del árbol y por medio de un número finito de rotaciones lo equilibre. Donde su número máximo de rotaciones para equilibrar un árbol AVL luego de una inserción es 2 y luego de una eliminación es log(n) donde n es el número de nodos.

En las secciones anteriores hemos ya descripto a grandes rasgos, cuál rotación usar en cada caso de desequilibrio. Esperamos que en el código siguiente se pueda formalizar lates ideas.

void balancear (struct Nodo \*\* t);

/\* Detecta y corrige por medio de un número finito de rotaciones

un desequilibrio en el árbol \*t. Dicho desequilibrio no debe

tener una diferencia de alturas de más de 2. \*/

void balancear (struct Nodo \*\* t)

{

if(!es\_vacio(\*t))

{

if (altura (izquierdo (\*t)) - altura (derecho (\*t)) == 2)

{ /\* desequilibrio hacia la izquierda! \*/

if (altura ((\*t)->izq->izq) >= altura ((\*t)->izq->der))

/\* desequilibrio simple hacia la izquierda \*/

rotar\_s (t, true);

else

/\* desequilibrio doble hacia la izquierda \*/

rotar\_d (t, true);

}

else if (altura (derecho (\*t)) - altura (izquierdo (\*t)) == 2)

{ /\* desequilibrio hacia la derecha! \*/

if (altura ((\*t)->der->der) >= altura ((\*t)->der->izq))

/\* desequilibrio simple hacia la izquierda \*/

rotar\_s (t, false);

else

/\* desequilibrio doble hacia la izquierda \*/

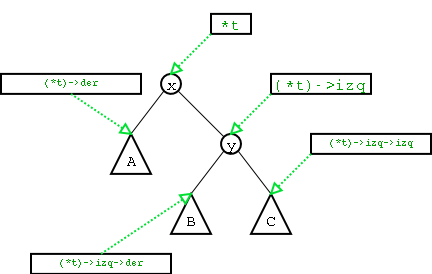
rotar\_d (t, false);

}

}

}

**Figura 12. Decidiendo qué clase de rotación aplicar para solucionar desequilibrio en el nodo.**



Como puede verse en el código, nos decidimos por una rotación simple izquierda si el subárbol más pesado de (\*t)->izq es el izquierdo o por una rotación doble izquierda si el subárbol más pesado de (\*t)->izq es el derecho.

**Inserción.**

Implementaremos la inserción de elementos de un árbol AVL de forma análoga a cómo lo haríamos para árboles binarios de búsqueda salvo que en cada recursión del algoritmo verificaremos y corregiremos el equilibrio del árbol. También es importante ir actualizando las alturas de cada nodo en cada recursión dado que las rotaciones, inserciones y eliminaciones pueden modificarlas.

Dado que ya vimos funciones tanto para balancear un árbol y para actualizar la altura de un nodo (ambas de tiempo de ejecución constante), estamos listos para implementar el algoritmo de inserción. Esperamos sea intuitivo.

void insertar (struct Nodo \*\* t, int x);

/\* inserta x en el árbol en un tiempo O(log(n)) peor caso. \*/

void insertar (struct \*\* t, int x)

{

if (es\_vacio (\*t))

\*t = hacer (x, vacio (), vacio ()); /\* altura actualizada

automáticamente \*/

else

{

if (x < raiz (\*t))

insertar (&(\*t)->izq, x);

else

insertar (&(\*t)->der, x);

balancear (t);

actualizar\_altura (\*t);

}

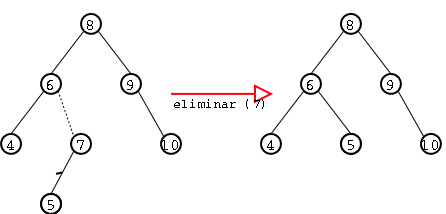
}

**Eliminación.**

La estrategia para diseñar el algoritmo de eliminación sobre árboles AVL es la misma que para la inserción: Se utiliza el mismo algoritmo que en árboles binarios de búsqueda, pero en cada recursión se detectan y corrigen errores por medio de balancear() y se actualiza la altura del nodo actual.

Recordamos un poco la idea del algoritmo de eliminación sobre árboles de búsqueda. Primero se recorre el árbol para detectar el nodo a eliminar. Una vez hecho esto, hay 3 casos a diferenciar por su complejidad:

* Si dicho nodo es hoja, procedemos a eliminarlos de inmediato, sin más.
* Si dicho nodo tiene un solo hijo, el nodo puede eliminarse después de ajustar un apuntador del padre para saltar el nodo. Tal y como se muestra a continuación:



**Figura 13. Eliminación de un nodo (7) con un solo hijo.**

* Si dicho nodo tiene dos hijos, el caso es un poco más complicado. Lo que se estila hacer (y que de hecho se hace en el algoritmo gracias a la función auxiliar eliminar\_min()) reemplazar el nodo actual por el menor nodo de su subárbol derecho (y luego eliminar éste).

void eliminar (struct Nodo \*\* t, int x);

/\* elimina x del árbol en un tiempo O(log(n)) peor caso.

Precondición: existe un nodo con valor x en el árbol

t. \*/

int eliminar\_min (struct Nodo \*\* t);

/\* Función auxiliar a eliminar(). Elimina el menor nodo del árbol

\*t devolviendo su contenido (el cual no se libera de

memoria). Se actualizan las alturas de los nodos.

Precondición: !es\_vacio(\*t) \*/

void eliminar (struct Nodo \*\* t, int x)

{

struct Nodo \*aux;

if (x < raiz (\*t))

eliminar (&(\*t)->izq, x);

else if (x > raiz (\*t))

eliminar (&(\*t)->der, x);

else /\* coincidencia! \*/

{

if (es\_vacio (izquierdo (\*t)) && es\_vacio (derecho (\*t)))

{/\* es una hoja \*/

free (\*t);

(\*t) = vacio();

}

else if (es\_vacio (izquierdo (\*t)))

{/\* subárbol izquierdo vacio \*/

aux = (\*t);

(\*t) = (\*t)->der;

free (aux);

}

else if (es\_vacio (derecho (\*t)))

{/\* subárbol derecho vacio \*/

aux = (\*t);

(\*t) = (\*t)->izq;

free (aux);

}

else /\* caso más complicado \*/

{

(\*t)->dato = eliminar\_min (&(\*t)->der);

}

}

balancear (t);

actualizar\_altura (\*t);

}

int eliminar\_min (struct Nodo \*\* t)

{

if (es\_vacio (\*t))

{

fprintf (stderr,

"No se respeta precondición de eliminar\_min()\n");

exit(0);

}

else

{

if (!es\_vacio (izquierdo (\*t)))

{

int x = eliminar\_min (&(\*t)->izq);

balancear (t);

actualizar\_altura (\*t);

return x;

}

else

{

Struct Nodo \*aux = (\*t);

int x = raiz (aux);

\*t = derecho (\*t);

free (aux);

balancear (t);

actualizar\_altura (\*t);

return x;

}

}

}

**Bibliografía:**

* Estructuras de datos y algoritmos, sección 4.4, pág. 114, Mark Allen Weiss.
* Data Structure Techniques, Standish, 1980, section 3.7.3.
* Handbook of Algorithms and Data Structures, Gonnet, 1984, section 3.4.1.
* <http://www.ibiblio.org/pub/linux/docs/LuCaS/Tutoriales/doc-programacion-arboles-avl/html/>