

# Desarrollo de software: ABB, AVL-tree y B-tree

Diego Campusano, Norton Irrarázabal, Nicolás García, Maykol González y Gerardo Carvajal.

26 de septiembre de 2018

# Índice

<b>1. Introducción</b>	<b>3</b>
<b>2. Analisis de un Árbol de Búsqueda Binaria (ABB)</b>	<b>4</b>
<b>3. Características</b>	<b>5</b>
<b>4. Tipos de Recorridos</b>	<b>6</b>
<b>5. Operaciones básicas</b>	<b>7</b>
<b>6. Aspectos técnicos del Desarrollo</b>	<b>10</b>
<b>7. Análisis del arbol AVL</b>	<b>12</b>
<b>8. Ejemplos</b>	<b>13</b>
<b>9. Implementación arbol AVL</b>	<b>14</b>
<b>10.Diagrama UML</b>	<b>15</b>
<b>11.Javadoc Generado sin errores</b>	<b>16</b>
<b>12.Muestra del Javadoc 1</b>	<b>17</b>
<b>13.Muestra del Javadoc 2</b>	<b>18</b>
<b>14.Muestra del Javadoc 3</b>	<b>19</b>
<b>15.Muestra del Javadoc 4</b>	<b>20</b>
<b>16.Diferencias entre algoritmos</b>	<b>21</b>
16.1. ABB y AVL-tree . . . . .	21
<b>17.Descripcion Arbol-B</b>	<b>22</b>
<b>18.Caracteristicas Arbol-B</b>	<b>22</b>
<b>19.Implementacion Arbol-B</b>	<b>23</b>
19.1. ABB y B-tree . . . . .	24
19.2. B-tree y AVL . . . . .	24
<b>20.Conclusión</b>	<b>26</b>

# 1. Introducción

En este presente trabajo daremos a conocer los algoritmos asociados a los ABB (Árbol de Búsqueda Binaria), AVL-tree (ABB balanceado) y B-tree(Árbol B).

Comenzando con el árbol de búsqueda binaria, haremos un profundo análisis a su efectiva capacidad para insertar, eliminar y realizar búsquedas en una cantidad no muy grande de datos, su deficiencia a la hora de realizar operaciones en numerosos datos y su costo relacionado, con esto podremos compararlo con los otros dos arboles, que poseen un cierto beneficio por su estructura, pero un costo adicional al tener más operaciones relacionadas, como rotaciones o divisiones internas. Estos últimos logran conseguir un balance que afecta positivamente al árbol, al momento de realizar determinadas operaciones

Ahora para los AVL, no se trata de árboles perfectamente equilibrados, pero sí lo son como para que su comportamiento sea lo bastante bueno como para usarlos, ya que los ABB no garantizan tiempos de búsqueda óptimos.

El algoritmo para mantener un árbol AVL equilibrado se basa en reequilibrados locales, de modo que no es necesario explorar todo el árbol después de cada inserción o borrado.

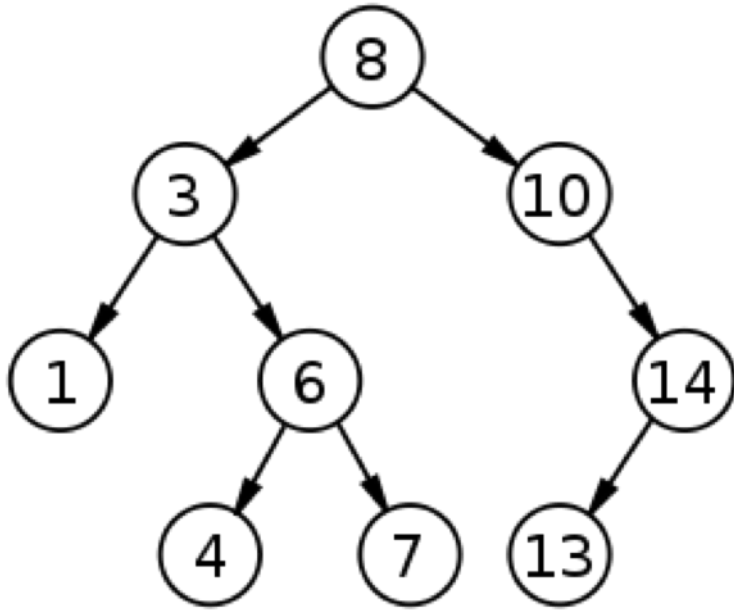
La búsqueda de estructura de datos más eficiente no cesa, y siempre existen oportunidades de encontrar nuevas estructura de datos con mejores prestaciones, de esta manera aparece el árbol B-Tree que es una estructura balanceada de búsqueda, diseñada para trabajar bien en discos u otros dispositivos de almacenamiento secundarios, puesto que ayudan a reducir las operaciones de I/O; muchas bases de datos utilizan este tipo de árbol o sus variantes para almacenar la información.

## 2. Analisis de un Árbol de Búsqueda Binaria (ABB)

- Definición: Un árbol de búsqueda binario ABB, es aquel que en cada nodo puede tener como mucho grado 2, es decir, un máximo de dos hijos. Los hijos suelen denominarse hijo izquierdo e hijo a la derecha, estableciéndose de esta forma un orden en el posicionamiento de los mismos. Se puede resumir en que es un árbol binario que cumple que el subárbol izquierdo de cualquier nodo (si no está vacío) contiene valores menores que el que contiene dicho nodo, y el subárbol derecho (si no está vacío) contiene valores mayores. Un árbol perfectamente equilibrado tiene el mismo número de nodos en el subárbol izquierdo que en el subárbol derecho. En un árbol binario equilibrado, el peor desempeño de insertar un dato es  $O(\log_2 n)$ , donde  $n$  es el número de nodos en el árbol. Para calcular la altura del árbol debemos calcular  $\log_2 n$ , y del mismo modo esta fórmula representa el número máximo de comparaciones que insertar necesitará hacer mientras busca el lugar apropiado para insertar un nodo nuevo.

### 3. Características

- Un árbol de búsqueda binaria es un árbol binario que almacena en cada uno una llave o valor
- El valor de la raíz es menor que los valores almacenados en el lado derecho
- El valor de la raíz es mayor que todos los valores almacenados en el lado izquierdo del nodo



## 4. Tipos de Recorridos

- Se puede hacer un recorrido de un árbol en profundidad o en anchura. Los recorridos en anchura son por niveles, se realiza horizontalmente desde la raíz a todos los hijos antes de pasar a la descendencia de alguno de los hijos. El coste de recorrer el ABB es  $O(n)$ , ya que se necesitan visitar todos los vértices. El recorrido en profundidad lleva al camino desde la raíz hacia el descendiente más lejano del primer hijo y luego continúa con el siguiente hijo. Como recorridos en profundidad tenemos inorden, preorden y postorden. Una propiedad de los ABB es que al hacer un recorrido en profundidad inorden obtenemos los elementos ordenados de forma ascendente.
- PreOrden: raíz-izquierda-derecha
- InOrden: izquierda-raíz-derecha
- PosOrden: izquierda-derecha-raíz

## 5. Operaciones básicas

- inserción: cuando se inserta un nuevo nodo en el árbol hay que tener en cuenta que cada nodo no puede tener más de dos hijos. La inserción es similar a la búsqueda y se puede dar una solución tanto iterativa como recursiva. Si tenemos inicialmente como parámetro un árbol vacío se crea un nuevo nodo como único contenido el elemento a insertar. Si no lo está, se comprueba si el elemento dado es menor que la raíz del árbol inicial con lo que se inserta en el subárbol izquierdo y si es mayor se inserta en el subárbol derecho.

```
TREE-INSERT(T, z)
1  y ← NIL
2  x ← root[T]
3  while x ≠ NIL
4      do y ← x
5          if key[z] < key[x]

6              then x ← left[x]
7              else x ← right[x]
8  p[z] ← y
9  if y = NIL
10     then root[T] ← z           ÷ Tree T was empty
11     else if key[z] < key[y]
12         then left[y] ← z
13         else right[y] ← z
```

- búsqueda: el algoritmo comprara el elemento a buscar con la raíz, si es menor continua la búsqueda por la rama izquierda, si es mayor continua por la derecha, este procedimiento se realiza recursivamente hasta que encuentre el nodo o hasta que llegue al final del árbol Cabe destacar que la búsqueda en este tipo de árboles es muy eficiente, representa una función logarítmica. El máximo número de comparaciones que necesitaríamos para saber si un elemento se encuentra en un árbol binario de búsqueda estaría entre  $\lceil \log_2(N+1) \rceil$  y  $N$ , siendo  $N$  el número de nodos.

```
TREE-SEARCH (x, k)
1  if x = NIL or k = key[x]
2    then return x
3  if k < key[x]
4    then return TREE-SEARCH(left[x], k)
5    else return TREE-SEARCH(right[x], k)
```



- **Borrar:** La operación de borrado no es tan sencilla como las de búsqueda e inserción. Existen varios casos a tener en consideración:
  - Borrar un nodo sin hijos o nodo hoja: simplemente se borra y se establece a nulo el apuntador de su padre.
  - Borrar un nodo con un subárbol hijo: se borra el nodo y se asigna su subárbol hijo como subárbol de su padre.
  - Borrar un nodo con dos subárboles hijo: la solución está en reemplazar el valor del nodo por el de su predecesor o por el de su sucesor en inorden y posteriormente borrar este nodo. Su predecesor en inorden será el nodo más a la derecha de su subárbol izquierdo (mayor nodo del subarbol izquierdo), y su sucesor el nodo más a la izquierda de su subárbol derecho (menor nodo del subarbol derecho). En la siguiente figura se muestra cómo existe la posibilidad de realizar cualquiera de ambos reemplazos:

```

TREE-DELETE(T, z)
1  if left[z] = NIL or right[z] = NIL
2      then y ← z
3      else y ← TREE-SUCCESSOR(z)
4  if left[y] ≠ NIL
5      then x ← left[y]
6      else x ← right[y]
7  if x ≠ NIL
8      then p[x] ← p[y]
9  if p[y] = NIL
10     then root[T] ← x
11     else if y = left[p[y]]
12         then left[p[y]] ← x
13         else right[p[y]] ← x
14 if y ≠ z
15     then key[z] ← key[y]
16     copy y's satellite data into z
17 return y

```

- Mínimo y máximo: Para encontrar el valor mínimo en un ABB se debe buscar en los hijos izquierdos hasta llegar al final del árbol, de modo contrario para encontrar un máximo se debe llegar al final del árbol por el lado derecho

```

TREE-MINIMUM (x)
1  while left[x] ≠ NIL
2      do x ← left[x]
3  return x

TREE-MAXIMUM(x)
1  while right[x] ≠ NIL
2      do x ← right[x]
3  return x

```

## 6. Aspectos técnicos del Desarrollo

- En primera instancia se pidió implementar un ABB en el lenguaje de programación Java, en este ABB se debe implementar un CRUD (excepto update), e ingresar la cantidad de 10.000 datos y ver cómo el árbol se comporta.

### 1. Estructura de Nodo implementada

```

class nodoArbol {

    arbolBusquedaBinaria hd;
    arbolBusquedaBinaria hi;
    private int numero;

    public nodoArbol(){
        hd = null;
        hi = null;
        numero = 0;
    }
    public int getNumero() {
        return numero;
    }
    public void setNumero(int numero) {
        this.numero = numero;
    }
}

```

### 2. Estructura de Árbol implementada

```

public class arbolBusquedaBinaria {
    private nodoArbol raiz;

    public arbolBusquedaBinaria() {
        @SuppressWarnings("unused")
        nodoArbol raiz = new nodoArbol();
    }
}

```

Esta estructura fue planteada de este modo ya que solo se usaron enteros como 'clave' en los árboles utilizados, a modo de hacer más simple la implementación y uso de estos.

- Diagrama de Clase



## 7. Análisis del árbol 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, es decir balanceado.

Recordamos que un árbol binario de búsqueda consiste en que 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.

El tiempo de las operaciones sobre un árbol binario de búsqueda son  $O(\log n)$  promedio, pero el peor caso es  $O(n)$ , donde  $n$  es el número de elementos.

La propiedad de equilibrio que debe cumplir un árbol para ser AVL asegura que la profundidad del árbol sea  $O(\log(n))$ , por lo que las operaciones sobre estas estructuras no deberán recorrer mucho para hallar el elemento deseado.

El tiempo de ejecución de las operaciones sobre estos árboles es, a lo sumo  $O(\log(n))$  en el peor caso, donde  $n$  es la cantidad de elementos del árbol. Sin embargo, y como era de esperarse, esta misma propiedad de equilibrio de los árboles AVL implica una dificultad a la hora de insertar o eliminar elementos. Ya que es probable que estas operaciones afecten la propiedad de equilibrio.

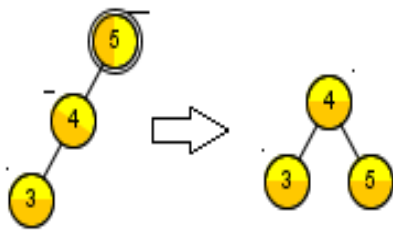
Por lo que se efectúan operaciones sobre el árbol conocidas como rotaciones, que nos ayudan a conservar el orden y a restaurar la propiedad de equilibrio de el árbol AVL.

*Factor de equilibrio o propiedad de equilibrio :  $FE = \text{altura subarbol derecho} - \text{altura sub arbol izquierdo}$ .*

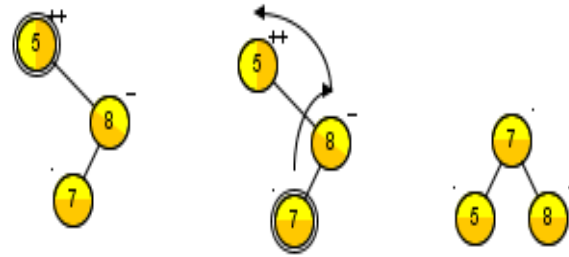
Se usan 4 tipos de rotaciones.

- Rotacion Simple a la derecha: Se usa cuando el subarbol izquierdo de un nodo sea 2 unidades mas alto que el derecho, es decir que este cargado a la izquierda.  $FE (= -2)$
- Rotacion Simple a la izquierda: Se usa cuando subarbol derecho de un nodo sea 2 unidades mas alto que el izquierdo, es decir que este cargado a la derecha. ( $FE=2$ )
- Rotacion Doble a la derecha: Si  $FE>1$  y su hijo derecho tiene signo -. Esto es el equivalente a i: aplicar rotacion simple a la derecha. ii: aplicar rotacion simple a la izquierda.
- Rotacion Doble a la izquierda: Si  $FE<-1$  y su hijo izquierdo tiene signo +. i: aplicar rotacion simple a la izquierda. ii: aplicar rotacion simple a la derecha.

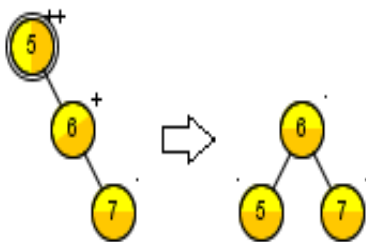
## 8. Ejemplos



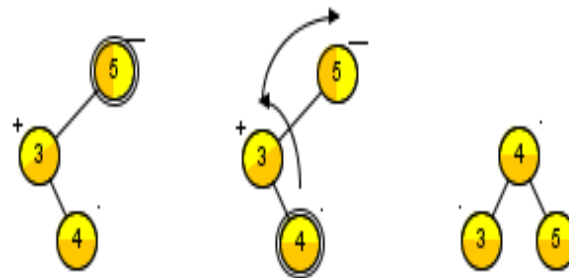
Rotación simple a la derecha.



Rotacion doble a la derecha.



Rotacion simple a la izquierda.



Rotacion doble a la izquierda.

## 9. Implementación arbol AVL

El arbol avl se elaboro en Eclipse Jee Oxygen es una plataforma de software compuesto por un conjunto de herramientas de programación de código abierto multiplataforma para desarrolladores java.

Pueden existir diversas formas de implementar un arbol AVL. Sin embargo, la implementación que yo hice para los nodos del arbol AVL es la siguiente:

```
public class Nodo_arbol_AVL {
    int Valor, FE;
    Nodo_arbol_AVL hijoizquierdo, hijoderecho;

    public Nodo_arbol_AVL (int Valor1) {
        this.Valor=Valor1;
        this.FE=0;
        this.hijoizquierdo=null;
        this.hijoderecho=null;
    }
}
```

Los nodos pueden tener muchos mas datos ya sean String, int, double etc. Sin embargo por simplicidad solo usaremos un dato de tipo entero, ahora si lo queremos con mas datos bastaria con incorporar otra variable.

Cada nodo tendra su propio factor de equilibrio, para poder evaluar el balance del arbol AVL. Ademas de un hijo izquierdo y un hijo derecho de su misma clase.

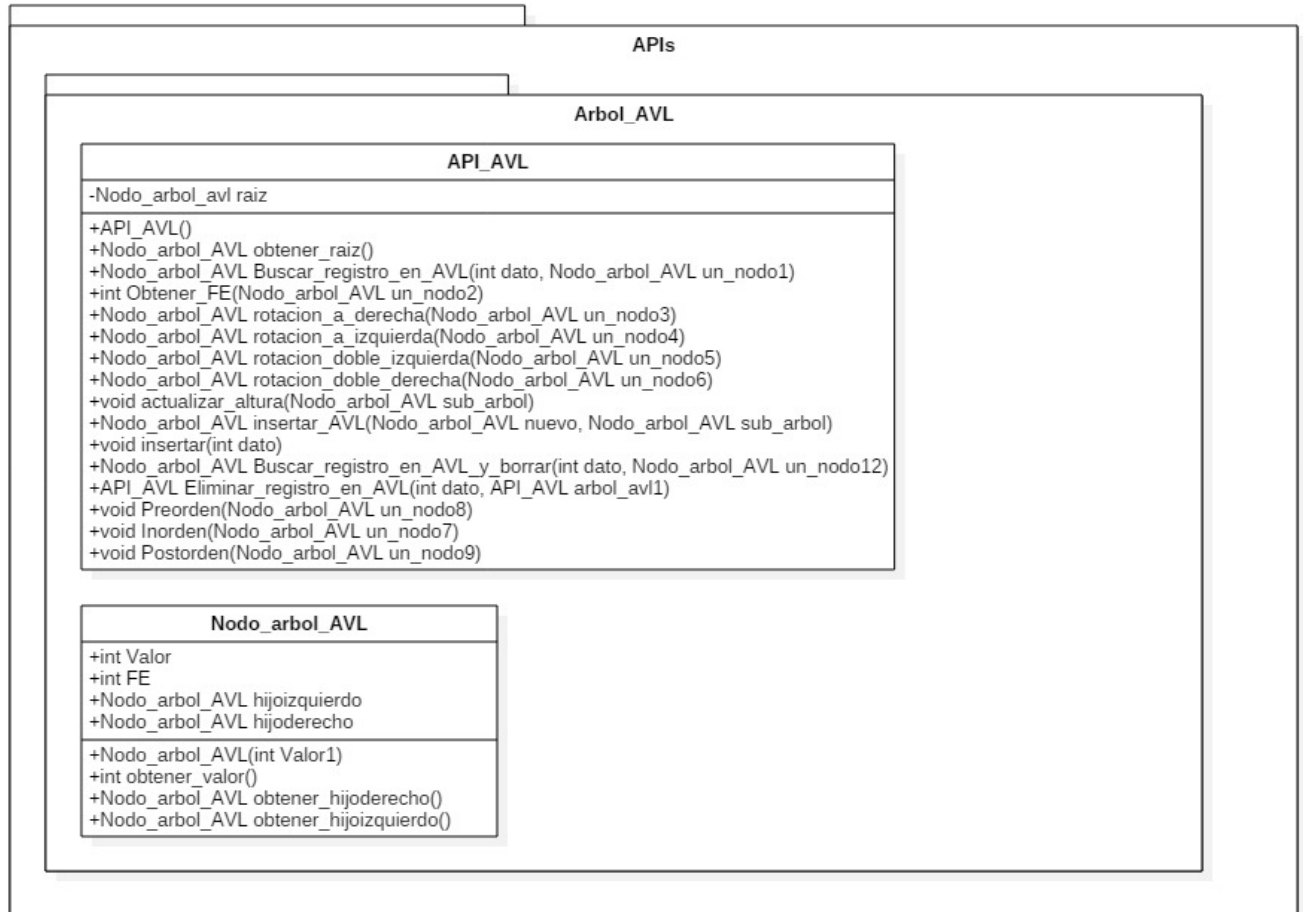
Ahora el arbol AVL lo defini como API\_AVL y este tendra solamente el nodo raiz.

```
public class API_AVL{
    private Nodo_arbol_AVL raiz;

    public API_AVL() {
        raiz=null;
    }
}
```

Ahora recordemos que cada una de estas clases tiene múltiples metodos y por consiguiente, incorpore diagramas UML y el Javadoc en donde podran encontrar todo el detalle.

## 10. Diagrama UML



## 11. Javadoc Generado sin errores

```
<terminated> Javadoc Generation
Loading source files for package APIs.Arbol_AVL...
Constructing Javadoc information...
Standard Doclet version 1.8.0_162
Building tree for all the packages and classes...
Generating C:\Users\Dante\eclipse-workspace\Tarea_2_daa\doc\APIs\Arbol_AVL\API_AVL.html...
Generating C:\Users\Dante\eclipse-workspace\Tarea_2_daa\doc\APIs\Arbol_AVL\Nodo_arbol_AVL.html...
Generating C:\Users\Dante\eclipse-workspace\Tarea_2_daa\doc\APIs\Arbol_AVL\package-frame.html...
Generating C:\Users\Dante\eclipse-workspace\Tarea_2_daa\doc\APIs\Arbol_AVL\package-summary.html...
Generating C:\Users\Dante\eclipse-workspace\Tarea_2_daa\doc\APIs\Arbol_AVL\package-tree.html...
Generating C:\Users\Dante\eclipse-workspace\Tarea_2_daa\doc\constant-values.html...
Generating C:\Users\Dante\eclipse-workspace\Tarea_2_daa\doc\APIs\Arbol_AVL\class-use\Nodo_arbol_AVL.html...
Generating C:\Users\Dante\eclipse-workspace\Tarea_2_daa\doc\APIs\Arbol_AVL\class-use\API_AVL.html...
Generating C:\Users\Dante\eclipse-workspace\Tarea_2_daa\doc\APIs\Arbol_AVL\package-use.html...
Building index for all the packages and classes...
Generating C:\Users\Dante\eclipse-workspace\Tarea_2_daa\doc\overview-tree.html...
Generating C:\Users\Dante\eclipse-workspace\Tarea_2_daa\doc\index-files\index-1.html...
Generating C:\Users\Dante\eclipse-workspace\Tarea_2_daa\doc\index-files\index-2.html...
Generating C:\Users\Dante\eclipse-workspace\Tarea_2_daa\doc\index-files\index-3.html...
Generating C:\Users\Dante\eclipse-workspace\Tarea_2_daa\doc\index-files\index-4.html...
Generating C:\Users\Dante\eclipse-workspace\Tarea_2_daa\doc\index-files\index-5.html...
Generating C:\Users\Dante\eclipse-workspace\Tarea_2_daa\doc\index-files\index-6.html...
Generating C:\Users\Dante\eclipse-workspace\Tarea_2_daa\doc\index-files\index-7.html...
Generating C:\Users\Dante\eclipse-workspace\Tarea_2_daa\doc\index-files\index-8.html...
Generating C:\Users\Dante\eclipse-workspace\Tarea_2_daa\doc\deprecated-list.html...
Building index for all classes...
Generating C:\Users\Dante\eclipse-workspace\Tarea_2_daa\doc\allclasses-frame.html...
Generating C:\Users\Dante\eclipse-workspace\Tarea_2_daa\doc\allclasses-noframe.html...
Generating C:\Users\Dante\eclipse-workspace\Tarea_2_daa\doc\index.html...
Generating C:\Users\Dante\eclipse-workspace\Tarea_2_daa\doc\help-doc.html...
```



## 12. Muestra del Javadoc 1

APIs.Arbol\_AVL

### Class API\_AVL

java.lang.Object  
APIs.Arbol\_AVL.API\_AVL

```
public class API_AVL
extends java.lang.Object
```

#### Constructor Summary

##### Constructors

##### Constructor and Description

API\_AVL()

#### Method Summary

##### All Methods

##### Instance Methods

##### Concrete Methods

Modifier and Type	Method and Description
void	actualizar_altura(Nodo_arbol_AVL sub_arbol) Actualizar altura
Nodo_arbol_AVL	Buscar_registro_en_AVL_y_borrar(int dato, Nodo_arbol_AVL un_nodo12) Buscar un registro y borrarlo, uso recursivo
Nodo_arbol_AVL	Buscar_registro_en_AVL(int dato, Nodo_arbol_AVL un_nodo1) Buscar y Leer un dato en el arbol AVL, uso recursivo
API_AVL	Eliminar_registro_en_AVL(int dato, API_AVL arbol_avl1) Eliminar registro en arbol AVL

## 13. Muestra del Javadoc 2

### *Method Detail*

#### obtener\_raiz

```
public Nodo_arbol_AVL obtener_raiz()
```

obtiene la raiz del arbol

Returns:

devuelve un nodo en este caso la raiz del arbol

#### Buscar\_registro\_en\_AVL

```
public Nodo_arbol_AVL Buscar_registro_en_AVL(int dato,  
                                              Nodo_arbol_AVL un_nodo1)
```

Buscar y Leer un dato en el arbol AVL, uso recursivo

Parameters:

dato - numero entero a buscar en el arbol

un\_nodo1 - un nodo en el arbol avl

Returns:

retorna un nodo

#### Obtener\_FE

```
public int Obtener_FE(Nodo_arbol_AVL un_nodo2)
```

Obtener factor de equilibrio

Parameters:

un\_nodo2 - un nodo de la clase Nodo\_arbol\_AVL

Returns:

retorna un entero para este caso el factor de equilibrio

## 14. Muestra del Javadoc 3

APIs.Arbol\_AVL

### Class Nodo\_arbol\_AVL

java.lang.Object  
APIs.Arbol\_AVL.Nodo\_arbol\_AVL

```
public class Nodo_arbol_AVL  
extends java.lang.Object
```

Nodo del arbol AVL

#### Constructor Summary

##### Constructors

##### Constructor and Description

Nodo_arbol_AVL(int Valor1)
parametros

#### Method Summary

##### All Methods

##### Instance Methods

##### Concrete Methods

Modifier and Type	Method and Description
Nodo_arbol_AVL	obtener_hijoderecho() obtener_hijoderecho
Nodo_arbol_AVL	obtener_hijoizquierdo() obtener_hijoizquierdo
int	obtener_valor() obtener_valor

## 15. Muestra del Javadoc 4

### *Method Detail*

#### **obtener\_valor**

```
public int obtener_valor()
```

obtener\_valor

Returns:

Devuelve el dato del nodo

#### **obtener\_hijoderecho**

```
public Nodo_arbol_AVL obtener_hijoderecho()
```

obtener\_hijoderecho

Returns:

devuelve un nodo

#### **obtener\_hijoizquierdo**

```
public Nodo_arbol_AVL obtener_hijoizquierdo()
```

obtener\_hijoizquierdo

Returns:

devuelve un nodo

## 16. Diferencias entre algoritmos

### 16.1. ABB y AVL-tree

Como sabemos, los árboles binarios de búsqueda son una estructura de datos que intenta conseguir mejor tiempo de acceso a los datos en las operaciones de búsqueda/recuperación, inserción o eliminación comparado con los tiempos en estructuras lineales como arreglos y listas. El acceso a un dato es proporcional a la altura del árbol, ya que su ubicación podría ser, en el peor de los casos, en una hoja. Por lo tanto, es deseable que el ABB tenga la menor altura posible; pero esto dependerá de la secuencia en que los datos se fueron insertando en el momento de la creación del mismo. De esta forma, en el peor de los casos, la búsqueda puede llegar a tener orden de complejidad  $O(n)$ , siendo que lo que buscamos es acercarnos a  $O(\log n)$ . La búsqueda de estructura de datos más eficiente no cesa, y siempre existen oportunidades de encontrar nuevas estructura de datos con mejores prestaciones, de esta manera aparecen el arboles B-Tree que es un árbol balanceado de búsqueda que más que todos son diseñados para trabajar bien en discos u otros dispositivos de almacenamiento secundarios, puesto que ayudan a reducir las operaciones de I/O; muchas bases de datos utilizan este tipo de árbol o sus variantes para almacenar la información.

## 17. Descripcion Arbol-B

Los arboles b disenados para funcionar bien en discos u otros accesos directos dispositivos de almacenamiento secundario. Los arboles B son similares a los arboles red-black, pero son mejores para minimizar las operaciones de E / S de disco. Muchos sistemas de bases de datos utilizan B-trees, o variantes de B-trees, para almacenar informacion.

La idea tras los arboles-B es que los nodos internos deben tener un numero variable de nodos hijo dentro de un rango predefinido. Cuando se inserta o se elimina un dato de la estructura, la cantidad de nodos hijo varía dentro de un nodo. Para que siga manteniendose el numero de nodos dentro del rango predefinido, los nodos internos se juntan o se parten. Dado que se permite un rango variable de nodos hijo, los arboles-B no necesitan Re balancearse tan frecuentemente como los arboles binarios de busqueda auto-balanceables. Pero, por otro lado, pueden desperdiciar memoria, porque los nodos no permanecen totalmente ocupados.

## 18. Caracteristicas Arbol-B

Los Arboles B deben cumplir las sigientes caracteristicas:

- Toda pagina tiene como maximo  $2n$  nodos.
- Toda pagina distinta de la raiz tiene como minimo  $n$  nodos.
- La raiz tiene como minimo un nodo.
- Todas las paginas hojas estan en el ultimo nivel.

Ademas de estas caracteristicas los arboles B tienen cumplir cierto orden:

- Los nodos dentro de una pagina mantienen un orden ascendente.
- Cada nodo es mayor que los nodos situados a su izquierda.
- Cada nodo es mayor que los nodos situados a su derecha.

## 19. Implementacion Arbol-B

En esta seccion, presentamos los detalles de algunas de las operaciones basicas del los arboles B como los es crear, Buscar, eliminar e Insertar:

- Crear Arbol B: Para construir un T arbol B, primero usamos B-TREE-CREATE para crear un nodo raiz vacio y luego agregamos nuevas claves.

```
B-TREE-CREATE(T)
1  x ← ALLOCATE-NODE()
2  leaf[x] ← TRUE
3  n[x] ← 0
4  DISK-WRITE(x)
5  root[T] ← x
```

- Buscar: Buscar un arbol B es muy parecido a buscar en un arbol binario de busqueda, excepto que en lugar de hacer un decision de bifurcacion binaria o "bidireccional".<sup>en</sup> cada nodo, hacemos una bifurcacion de multiples vias decision segun el numero de hijos del nodo.

```
B-TREE-SEARCH(x, k)
1  i ← 1
2  while i ≤ n[x] and k > keyi[x]
3      do i ← i + 1
4  if i ≤ n[x] and k = keyi[x]
5      then return (x, i)
6  if leaf [x]
7      then return NIL
8  else DISK-READ(ci[x])
9      return B-TREE-SEARCH(ci[x], k)
```

Usando un procedimiento de busqueda lineal, las lineas 1-3 encuentran el indice mas pequeno i tal que  $k \leq \text{key}_i(x)$ , o de lo contrario, establecen i a  $n[x] + 1$ . Las lineas 4-5 comprueban si ahora hemos descubierto la clave, volviendo si tenemos Las lineas 6-9 terminan la busqueda sin exito (si x es una hoja) en el busque el subarbol apropiado de x, despues de realizar el DISK-READ necesario en ese hijo

- Insertar: Las inserciones se hacen en los nodos hoja.
  - 1. Realizando una busqueda en el arbol, se halla el nodo hoja en el cual deberia ubicarse el nuevo elemento.
  - 2. Si el nodo hoja tiene menos elementos que el maximo numero de elementos legales, entonces hay lugar para uno mas. Inserte el nuevo elemento en el nodo, respetando el orden de los elementos.
  - 3. De otra forma, el nodo debe ser dividido en dos nodos. La division se realiza de la siguiente manera:
    - Se escoge el valor medio entre los elementos del nodo y el nuevo elemento.

- Los valores menores que el valor medio se colocan en el nuevo nodo izquierdo, y los valores mayores que el valor medio se colocan en el nuevo nodo derecho; el valor medio actúa como valor separador.
- El valor separador se debe colocar en el nodo padre, lo que puede provocar que el padre sea dividido en dos, y así sucesivamente.

```

B-TREE-INSERT( $T, k$ )
1   $r \leftarrow \text{root}[T]$ 
2  if  $n[r] = 2t - 1$ 
3      then  $s \leftarrow \text{ALLOCATE-NODE}()$ 
4           $\text{root}[T] \leftarrow s$ 
5           $\text{leaf}[s] \leftarrow \text{FALSE}$ 
6           $n[s] \leftarrow 0$ 
7           $c_1[s] \leftarrow r$ 
8          B-TREE-SPLIT-CHILD( $s, 1, r$ )
9          B-TREE-INSERT-NONFULL( $s, k$ )
10 else B-TREE-INSERT-NONFULL( $r, k$ )  Insercion normal

```

```

B-TREE-INSERT-NONFULL( $x, k$ )
1   $i \leftarrow n[x]$ 
2  if  $\text{leaf}[x]$ 
3      then while  $i \geq 1$  and  $k < \text{key}_i[x]$ 
4          do  $\text{key}_{i+1}[x] \leftarrow \text{key}_i[x]$ 
5               $i \leftarrow i - 1$ 
6           $\text{key}_{i+1}[x] \leftarrow k$ 
7           $n[x] \leftarrow n[x] + 1$ 
8          DISK-WRITE( $x$ )
9  else while  $i \geq 1$  and  $k < \text{key}_i[x]$ 
10     do  $i \leftarrow i - 1$ 
11      $i \leftarrow i + 1$ 
12     DISK-READ( $c_i[x]$ )
13     if  $n[c_i[x]] = 2t - 1$ 
14         then B-TREE-SPLIT-CHILD( $x, i, c_i[x]$ )
15             if  $k > \text{key}_i[x]$ 
16                 then  $i \leftarrow i + 1$ 
17     B-TREE-INSERT-NONFULL( $c_i[x], k$ )  Insercion con division de nodo

```

- Eliminar: La eliminación de un B-tree es análoga a la inserción, pero un poco más complicada, porque una clave puede eliminarse de cualquier nodo, no solo una hoja, y la eliminación de un nodo interno requiere que los hijos del nodo se reorganicen. Al igual que en la inserción, debemos protegernos contra la eliminación produciendo un árbol cuya estructura viola las propiedades del árbol B. Así como tenemos que asegurarnos de que un nodo no se hace demasiado grande debido a la inserción, debemos asegurarnos de que un nodo no se vuelva demasiado pequeño durante la eliminación (excepto que la raíz tiene permitido tener menos que el número mínimo  $t - 1$  de teclas, aunque no está permitido tener más que el número máximo  $2t - 1$  de teclas).

## 19.1. ABB y B-tree

A diferencia del ABB, el árbol B posee múltiples caminos, ya que cada nodo puede tener varios hijos según el grado del árbol, y es auto-balanceable por lo que todas sus hojas están en el mismo nivel, esto implica que en comparación las búsquedas son mucho más directas. Cabe recalcar también que las hojas contienen todas las claves en orden, por lo que las búsquedas son realmente efectivas. donde está mucho más dividido la zona de claves en donde queramos buscar.

## 19.2. B-tree y AVL

La principal diferencia entre el ABB y el B-tree, es que este último realiza una menor cantidad de búsqueda para poder llegar a la clave solicitada. Los árboles B tienen ventajas sustanciales sobre otras implementaciones cuando el tiempo de acceso a los nodos excede al tiempo de acceso entre nodos. Este caso se da usualmente cuando



los nodos se encuentran en dispositivos de almacenamiento secundario como los discos rígidos. Al maximizar el número de nodos hijo de cada nodo interno, la altura del árbol decrece, las operaciones para balancearlo se reducen, y aumenta la eficiencia. Usualmente este valor se coloca de forma tal que cada nodo ocupe un bloque de disco, o un tamaño análogo en el dispositivo. Depende de como estén implementados estos árboles para decidir cual funciona mejor a la hora realizar operaciones

- En el peor de los casos, la altura de un árbol-B es:
- $\log_m N$
- En el mejor de los casos, la altura de un árbol-B es:
- $N$

Donde  $M$  es el número máximo de hijos que puede tener un nodo.

## 20. Conclusión

Para culminar nuestro trabajo debemos tener en cuenta que un árbol como estructura de datos nos permite almacenar una cantidad significativa de datos de forma ordenada. Un árbol se representa con un conjunto de nodos entrelazados entre sí por medio de ramas, debemos tener en cuenta que el nodo base es único, y se le denomina raíz. En un árbol un padre puede tener varios hijos pero un hijo solo puede tener un padre, desde la raíz se puede llegar a cualquier nodo progresando por las ramas y atravesando los sucesivos niveles estableciendo así un camino.

Los arboles nacen de la necesidad de organizar la información de manera óptima, así que su importancia no es menor, ya que la velocidad de búsqueda, inserción y borrado depende mucho de una buena implementación de estas estructuras de datos. ya que para cada problemática de esta índole, siempre habrá un árbol que sea más adecuado que otro.

## Referencias

- [1] Árboles AVL: [rua.ua.es/dspace/handle/10045/16037](http://rua.ua.es/dspace/handle/10045/16037)
- [2] Estructuras de datos : Libro CLRS
- [3] Árboles binarios de búsqueda: [decsai.ugr.es/jfv/ed1/tedi/cdrom/docs/arb\\_BB.htm](http://decsai.ugr.es/jfv/ed1/tedi/cdrom/docs/arb_BB.htm)
- [4] Árboles B: [computersciencesmeraldas.blogspot.cl/2016/04/btreealgomassobrearboles.html](http://computersciencesmeraldas.blogspot.cl/2016/04/btreealgomassobrearboles.html)