# Documentación sobre método que obtiene la profundidad de un árbol binario.

Por Oscar Alberto Salgado Cárdenas
oscasc@gmail.com

La construcción del árbol binario, reside en tres clases existentes en el paquete *com.neixar.osalgado.dtStructures*

- ## Arbol.java
  - Clase de tipo ***public***. Que permite la construcción de una estructura de árbol tipo binario y recibe (a partir de esta nueva versión) la inserción de números de tipo entero. Este árbol se conformará dependiendo el órden de inserción de los números enteros para determinar si el nuevo elemento se insertará a la izquierda o a la derecha.
  - Su constructor carece de parámetros para su inicialización.
  - **Métodos** (públicos)
    - void add(int): Ingresa un nuevo elemento (nodo) al árbol.
    - int getDepth(): Retorna la profundidad máxima del arbol actualmente conformado.
      - Retorno del valor 0: Significa que el árbol, carece de nodos y elemento raíz.
      - Retorno del valor 1: Significa que el árbol, sólo cuenta con el nodo raíz.
      - Retorno de un valor mayor a 1: Significa que el árbol, cuenta con más nodos, además del raíz.
    - Nodo getNodo(int): Es un método no implementado, pero podría considerarse su desarrollo futuro, para navegar en el árbol, buscando el valor indicado y retornando el nodo correspondiente, (en caso de encontrarlo).
  - **Métodos** (privados)
    - void add(Nodo, Nodo) Es utilizado por el método getNodo(int), para realizar la insersión recursiva de los valores que ingresan al árbol.
    - int getDepth(Nodo, int) Es utilizado por el método getDepth(), para realizar la contabilización de niveles de profundidad, de manera recursiva.
- ## Nodo.java
  - Clase de tipo ***protected***. Es el elemento básico del árbol. Cada nodo conectará a su vez con uno o dos nodos más, que llamaremos “nodos hijos”. El árbol, comenzará con un primer nodo al que llamaremos nodo raíz (root). Cada nodo constituye la siguiente información:
    - Contenido: Almacena números los valores de tipo entero, que ingresan al

árbol. Dependiendo de su valor y en comparación con el valor del nodo padre al que se agregarán, se ingresará como el nodo hijo colocado a la izquierda o a la derecha. En caso de ingresar un valor ya existente al árbol, éste se ignorará sin afectar al árbol.

- Nodo izquierdo: Almacena un objeto de tipo nodo, para conectar con los nuevos valores que igresarán al árbol. Mediante una comparación de valores, si se trata de un valor menor que el del nodo padre, se coloca en el nodo izquierdo.
- Nodo derecho: Almacena un objeto de tipo nodo, para conectar con los nuevos valores que igresarán al árbol. Mediante una comparación de valores, si se trata de un valor mayor que el del nodo padre, se coloca en el nodo deerecho.
- Nodo padre: Almacena un objeto de tipo nodo. Conserva la referencia del nodo que previamente lo referenció. (El nodo padre). No tiene utilización para esta versión, pero puede utilizarse para hacer recorridos en el árbol partiendo de un nodo hijo en dirección de los consecutivos padres.

- Métodos protegidos
  - Nodo getLeft()  Retorna el Nodo hijo izquierdo del nodo actual.
  - Nodo getRight() Retorna el Nodo hijo derecho del nodo actual.
  - void setLeft(Nodo) Asigna como nodo hijo izquierdo al nodo actual.
  - void setRight()  Asigna como nodo hijo derecho al nodo actual.
  - int getContenido() Extrael el valor, contenido en el nodo actual.
  - void setContenido(int) Ingresa el valor, contenido al nodo actual.
  - void setPadre(Nodo) Establece la relación de un nodo, con su nodo padre
  - Nodo getPadre() Obtiene el nodo Padre relacionado al nodo actual.

- **ArbolI.java**
  - Interface que será utilizada por la clase Arbol.java y que define los siguientes métodos:
    - void add(int)
    - int getDepth()
    - Nodo getNodo(int)

### Construcción del árbol

La conformación del árbol dependerá de el órden de ingreso de sus datos, dado que éstos se acomodan entre nodos hijos, mediante una evaluación previa para con el valor del nodo padre.

El procedimiento de inserción será de la siguiente forma:

1. Ingresa un elemento:

2. ¿El nodo root, existe? Si no existe, el nuevo elemento es el Root; FIN

3. En caso contrario evaluamos:

    a) ¿El valor del nuevo elemento es menor que el valor del nodo (ahora llamado padre)?

        i. Si es sí:

            1. ¿El nodo hijo izquierdo, existe?

                a) Si es no: Insertamos como nodo hijo izquierdo

                b) Si ya existe, ahora ese nodo hijo izquierdo será considerado como el padre y repetimos la operación desde el punto 3a (recursivo)

        ii. Si es mayor:

            1. ¿El nodo hijo derecho, existe?

                a) Si no existe: Insertamos como nodo hijo derecho

                b) Si ya existe, ahora ese nodo hijo derecho será considerado como el padre y repetimos la operación desde el punto 3a (recursivo)

        iii. Si es igual : NO insertamos nada y FIN

### Método getDepth()

Este es un método igualmente recursivo que comienza primeramente, viajando en dirección, primero a los nodos hijos izquierdo (aunque lo mismo hubiese sido si primero lo hiciera por el lado derecho) y en cada iteración aumenta un contador, llamado currDepth. Cuando el No encuentra más nodos hijos izquierdo qué visitar, comienza a visitar los nodos hijos derechos.

Cabe señalar que en cada iteración, al término de la navegación por la izquierda y la derecha, se evalúa la cantidad de niveles de profundidad que cada rama viajó y nos quedaremos con el mayor valor entre ambos. Dicho valor, será el que se burbujeará hasta la anterior recursión. (el ciclo recursivo que lo llamó).

## Ejercicios de prueba

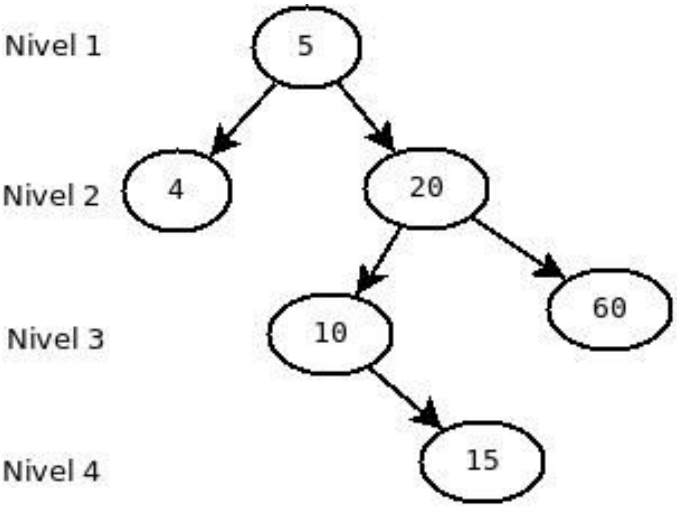
Se realizaron pruebas conformando listas de elementos con valores aleatorios (previamente definidos) a los que se conformó (a lápiz y papel) su árbol binario, previendo de esta manera y a medida como se fuesen ingresando los valores; el

programa (en la clase appMaiin.java que contiene el método main()) mostraría en cada iteración: inserción de valor y obtención del valor getDepth(); el comportamiento predicho:

## Ejemplo 1

Lista de elementos: `{5,,4,20,10,60,15}`

Crearía este tipo de árbol:

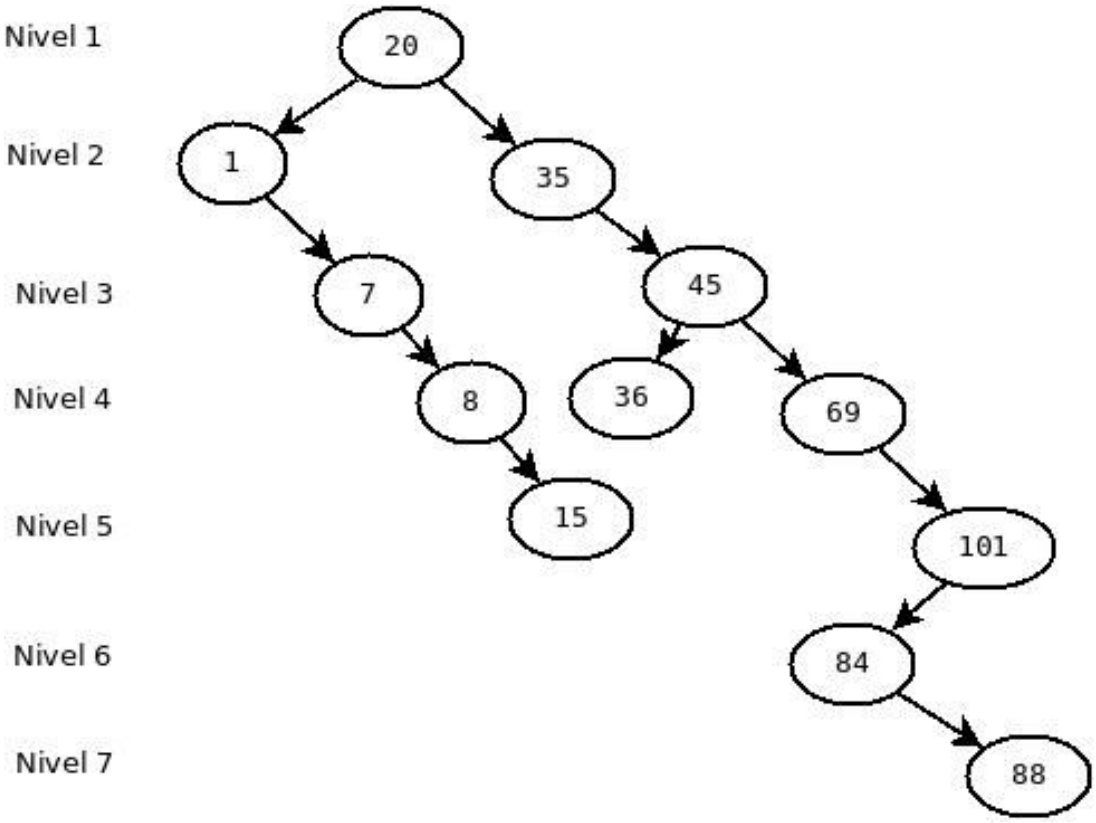



La salida del sistema es esta:

```
con x = 5 ::Profundidad actual:1
con x = 4 ::Profundidad actual:2
con x = 20 ::Profundidad actual:2
con x = 10 ::Profundidad actual:3
con x = 60 ::Profundidad actual:3
con x = 15 ::Profundidad actual:4
Fin
```

## Ejemplo 2

Lista de elementos: `{20,1,35,7,8,45,1,69,101,84,15,36,88}`

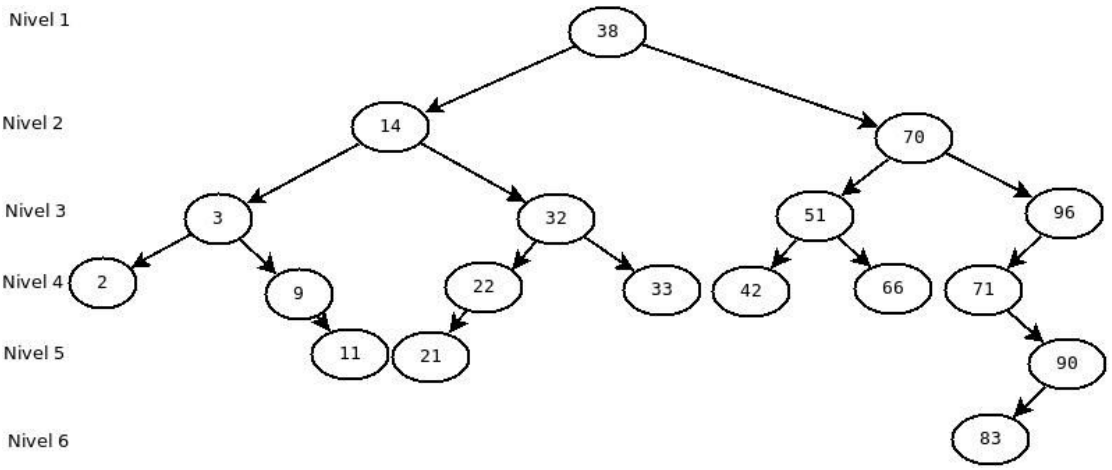



```
con x = 20 ::Profundidad actual:1
con x = 1 ::Profundidad actual:2
con x = 35 ::Profundidad actual:2
con x = 7 ::Profundidad actual:3
con x = 8 ::Profundidad actual:4
con x = 45 ::Profundidad actual:4
con x = 1 ::Profundidad actual:4
con x = 69 ::Profundidad actual:4
con x = 101 ::Profundidad actual:5
con x = 84 ::Profundidad actual:6
con x = 15 ::Profundidad actual:6
con x = 36 ::Profundidad actual:6
con x = 88 ::Profundidad actual:7
Fin
```

Ejemplo 3

Lista de elementos:

```
{ 38, 70, 14, 32, 96, 51, 3, 9, 22, 71, 66, 42, 11, 21, 9, 2, 90, 83,
42, 33 }
```




```
con x = 14 ::Profundidad actual:2
con x = 32 ::Profundidad actual:3
con x = 96 ::Profundidad actual:3
con x = 51 ::Profundidad actual:3
con x = 3 ::Profundidad actual:3
con x = 9 ::Profundidad actual:4
con x = 22 ::Profundidad actual:4
con x = 71 ::Profundidad actual:4
con x = 66 ::Profundidad actual:4
con x = 42 ::Profundidad actual:4
con x = 11 ::Profundidad actual:5
con x = 21 ::Profundidad actual:5
con x = 9 ::Profundidad actual:5
con x = 2 ::Profundidad actual:5
con x = 90 ::Profundidad actual:5
con x = 83 ::Profundidad actual:6
con x = 42 ::Profundidad actual:6
con x = 33 ::Profundidad actual:6
Fin
```

Cabe la observación: Que al momento de ingresar el elemento 83, éste ocupa la profundidad número 6 (que es la máxima). Luego de éste elemento, aún se insertarán los valores 42 y 33 que ocuparán (dentro del árbol) la profundidad del nivel 4, Sin embargo, dado que ya existe una profundidad máxima igual 6, el método reporta ese número y no el nivel 4 que es donde se insertaron estos dos últimos elementos. El métido, hace tal cual lo que se espera de él.