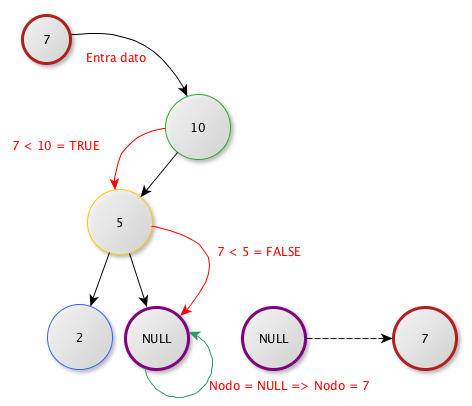
* **Hijo**: Es aquel nodo que siempre va a tener un nodo antecesor o padre, son aquellos que se encuentran en el mismo nivel
* **Padre**: Es aquel que tiene hijos y también puede tener o no antecesores.
* **Hermano**: Dos nodos son hermanos si son apuntados por el mismo nodo, es decir si tienen el mismo padre.
* **Raíz**: Es el nodo principal de un árbol y no tiene antecesores.
* **Hoja** o terminal: Son aquellos nodos que no tienen hijos o también los nodos finales de un árbol.
* **Interior**: Se dice que un nodo es interior si no es raíz ni hoja.
* **Nivel de un nodo**: Se dice que el nivel de un nodo es el numero de arcos que deben ser recorridos, partiendo de la raíz para llegar hasta el.
* **Altura del árbol**: Se dice que la altura de un árbol es el máximo de los niveles considerando todos sus nodos.
* **Grado de un nodo**: se dice que el grado de un nodo es el número de hijos que tiene dicho nodo.

La información se ingresa de forma ordenada esto se resuelve de forma muy sencilla con estos pasos:

1. Se toma el dato a ingresar X
2. Partiendo de la raíz preguntamos: Nodo == null ( o no existe ) ?
3. En caso afirmativo X pasa a ocupar el lugar del nodo y ya hemos ingresado nuestro primer dato.
4. En caso negativo preguntamos: X < Nodo
5. En caso de ser menor pasamos al Nodo de la IZQUIERDA del que acabamos de preguntar y repetimos desde el paso 2 partiendo del Nodo al que acabamos de visitar
6. En caso de ser mayor pasamos al Nodo de la DERECHA y tal cual hicimos con el caso anterior repetimos desde el paso 2 partiendo de este nuevo Nodo.

Nos daremos cuenta de que es un proceso *RECURSIVO*en el cual al final por más grande que sea el árbol el dato a entrar ocupará un lugar, vamos a ejemplificar lo ya mencionado con una imagen:



**Vamos a programarlo**

Todo lo dicho anteriormente, vamos a programarlo ahora usando POO con java (para que sea más fácil de entender).

Comenzamos con la abstracción de la información, tenemos que un árbol binario está compuesto por la raíz y sus nodos hijos, de la misma forma que la misma raíz no es más que otro nodo, partiendo de esto entonces crearemos 2 clases:

(Quiero comentar antes de esto que, aunque no soy muy partidario de escribir los nombres de los métodos y clases en español, por cuestiones de entendimiento para muchas personas que lleguen a esta información, los pondré de esa forma).

* Arbol
* Nodo

**Nodo.java**

public class Nodo {

/\* Declaraciones de variables \*/

private int valor;

private Nodo padre;

private Nodo hojaIzquierda;

private Nodo hojaDerecha;

/\* Constructor \*/

public Nodo(int valor) {

this.valor = valor;

}

/\* Setters y Getters \*/

public void setValor(int valor) {

this.valor = valor;

}

public int getValor() {

return valor;

}

public Nodo getPadre() {

return padre;

}

public void setPadre(Nodo padre) {

this.padre = padre;

}

public Nodo getHojaIzquierda() {

return hojaIzquierda;

}

public void setHojaIzquierda(Nodo hojaIzquierda) {

this.hojaIzquierda = hojaIzquierda;

}

public Nodo getHojaDerecha() {

return hojaDerecha;

}

public void setHojaDerecha(Nodo hojaDerecha) {

this.hojaDerecha = hojaDerecha;

}

}

Vamos a revisar un poco más de cerca el código.

Es una clase muy sencilla, como podemos ver únicamente tiene 4 atributos (variables) fáciles de entender:

* valor
* hojaIzquierda
* hojaDerecha

**valor** es el dato almacenado del propio nodo, mientras que como recordaremos cada nodo puede tener máximo 2 hijos uno a la izquierda y otro a la derecha, donde todos los nodos a su izquierda son menores a el mismo y todos los nodos a su derecha son mayores a el mismo, pero que comienzan como nulos ya que al crear un nuevo nodo, este no tiene hijos.

Hay que tener cuidado con setValor( int valor ) ¿Porque? pues bien, no podemos actualizar el valor al aire ya que perderíamos el orden, por lo que tenemos un contructor:

/\* Constructor \*/

public Nodo(int valor) {

this.valor = valor;

}

a partir del cual se ingresa el valor del nodo al crear una instancia del mismo:

Nodo nodo = new Nodo( 1 );

Ya tenemos nuestra clase nodo lista, ahora pasemos con la clase Arbol.java que es aún más sencilla ( en definición ) que Nodo:

**Arbol.java**

public class Arbol {

/\* Atributos \*/

private Nodo raiz;

/\* Contructories \*/

public Arbol( int valor ) {

this.raiz = new Nodo( valor );

}

public Arbol( Nodo raiz ) {

this.raiz = raiz;

}

/\* Setters y Getters \*/

public Nodo getRaiz() {

return raiz;

}

public void setRaiz(Nodo raiz) {

this.raiz = raiz;

}

}

Prácticamente por ahora lo importante es su atributo «raiz» del tipo Nodo, el cual representará a todo nuestro árbol, adicionalmente he creado constructores que pueden ser de ayuda al momento de inicializar el árbol para crear la raíz de paso.

Hay que agregar métodos a nuestra clase de Arbol para poder ingresar nuevos nodos:

* addNodo
* removeNodo
* recorridoPreorden
* recorridoInorden
* recorridoPostorden

**Programando las funciones del árbol**

En esta primera parte quiero abarcar al menos hasta el ingreso de nuevos nodos, comencemos entonces.

La modificación la haremos en nuestra clase Arbol que es la que llevará estas funciones de control, recordemos nuestro algoritmo de inserción que planteamos anteriormente:

1. Se toma el dato a ingresar X
2. Partiendo de la raíz preguntamos: Nodo == null ( o no existe ) ?
3. En caso afirmativo X pasa a ocupar el lugar del nodo y ya hemos ingresado nuestro primer dato.
4. En caso negativo preguntamos: X < Nodo
5. En caso de ser menor pasamos al Nodo de la IZQUIERDA del que acabamos de preguntar y repetimos desde el paso 2 partiendo del Nodo al que acabamos de visitar
6. En caso de ser mayor pasamos al Nodo de la DERECHA y tal cual hicimos con el caso anterior repetimos desde el paso 2 partiendo de este nuevo Nodo.

Entonces definamos nuestra función:

private void private void addNodo( Nodo nodo, Nodo raiz ) {

/\* 2.- Partiendo de la raíz preguntamos: Nodo == null ( o no existe ) ? \*/

if ( raiz == null ) {

\* 3.- En caso afirmativo X pasa a ocupar el lugar del nodo y ya

\* hemos ingresado nuestro primer dato.

this.setRaiz(nodo);

}

else {

/\* 4.- En caso negativo preguntamos: X < Nodo \*/

if ( nodo.getValor() <= raiz.getValor() ) {

/\*

\* 5.- En caso de ser menor pasamos al Nodo de la IZQUIERDA del

\* que acabamos de preguntar y repetimos desde el paso 2

\* partiendo del Nodo al que acabamos de visitar

\*/

if (raiz.getHojaIzquierda() == null) {

raiz.setHojaIzquierda(nodo);

}

else {

addNodo( nodo , raiz.getHojaIzquierda() );

}

}

else {

/\*

\* 6.- En caso de ser mayor pasamos al Nodo de la DERECHA y tal

\* cual hicimos con el caso anterior repetimos desde el paso 2

\* partiendo de este nuevo Nodo.

\*/

if (raiz.getHojaDerecha() == null) {

raiz.setHojaDerecha(nodo);

}

else {

addNodo( nodo, raiz.getHojaDerecha() );

}

}

}

}

public void addNodo( Nodo nodo ) {

this.addNodo( nodo , this.raiz );

}

Podemos ver que he agregado 2 funciones para insertar un nuevo nodo, pero solo 1 de ellas es vista desde fuera, la que nosotros llamaremos, esto es porque la función que  usamos realmente para insertar (llamada dentro del método que vemos desde fuera de la clase [el método público, para quien ande perdido]) necesita una referencia de la «raíz» a partir de que nodo va a hacer las comprobaciones, para que al usar la recursividad esta raíz pueda ir cambiando (de acuerdo al algoritmo) y seguir probando y comparando nodos.

# [**Búsqueda binaria en java**](http://codigomaldito.blogspot.com/2007/11/bsqueda-binaria-en-java.html)

Implementación del algoritmo de búsqueda binaria de manera no recursiva en Java.

Se utiliza una función estática de la clase BusquedaAlgoritmo.

Recordar que para que funcione correctamente los valores del arreglo deben estar ordenados.

**class BusquedaAlgoritmo** {

**public static int buscar( int [] arreglo, int dato)** {

int inicio = 0;

int fin = arreglo.length - 1;

int pos;

while (inicio <= fin) {

pos = (inicio+fin) / 2;

if ( arreglo[pos] == dato )

return pos;

else if ( arreglo[pos] < dato ) {

inicio = pos+1;

} else {

fin = pos-1;

}

}

return -1;

}

}

**public class BusquedaBinaria** {

**public static void main (String args[])** {

// Llenar arreglo

int [] edades = new int [35];

for (int i = 0; i < edades.length ; i++)

edades[i] = i\*i ;

// Mostrar arreglo.

for (int i = 0; i < edades.length ; i++)

System.out.println ( "edades["+i+"]: "+ edades[i]);

int resultado = BusquedaAlgoritmo.buscar(edades, 9);

if (resultado != -1) {

System.out.println ( "Encontrado en: "+ resultado);

} else {

System.out.println ( "El dato no se encuentra en el arreglo, o el arreglo no estÃ¡ ordenado." );

}

}

}

**Recorridos de Árboles**

**Preorden:**

1. Visitar la **Raíz**
2. Recorrer el subarbol **izquierdo**
3. Recorrer el subarbol **derecho**

**Inorden:**

1. Recorrer el subarbol **izquierdo**
2. Visitar la **raíz**
3. Recorrer el subarbol **derecho**

**Postorden:**

1. Recorrer el subarbol **izquierdo**
2. Recorrer el subarbol **derecho**
3. Visitar la **raíz**

# **Ejemplo de Código de Árboles en Java**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74 | package pruebaarbol;  import pruebaarbol.model.Nodo;    /\*\*  \*  \* @author Daniel  \*/  public class Pruebaarbol {      /\*\*    \* @param args the command line arguments    \*/    public static void main(String[] args) {        //Se asignan los valores a los nodos = 1,2,3,4,5 y 6 dentro del arbol      //Nodo raiz = 1      Nodo raiz = new Nodo(1);        //Nodo2 Izquierdo = 2      Nodo nodo2 = new Nodo(2);        //Nodo3 Derecho = 3      Nodo nodo3 = new Nodo(3);        //Se asigna el valor 6 al nodo que sera hijo del nodo 3 a la derecha      nodo3.setNodoDerecho(new Nodo(6));        //Se asigna el valor 5 al nodo que sera hijo del nodo 3 a la izquierda      nodo3.setNodoIzquierdo(new Nodo(5));        //Se asigna el valor 4 al nodo que sera hijo del nodo 4 a la izquierda      nodo2.setNodoIzquierdo(new Nodo(4));        //Se crean el Nodo 2 a la izquierda y el Nodo 3 a la derecha de la raiz      raiz.setNodoIzquierdo(nodo2);      raiz.setNodoDerecho(nodo3);        //Resultado en pantalla      System.out.println("Recorrido Preorden: ");      preOrden(raiz);      System.out.println("Recorrido Inorden: ");      inorden(raiz);      System.out.println("Recorrido PostOrden: ");      posOrden(raiz);    }      //Metodo Preorden    private static void preOrden(Nodo raiz) {      if (raiz != null) {        System.out.print(raiz.getDato() + " - ");        preOrden(raiz.getNodoIzquierdo());        preOrden(raiz.getNodoDerecho());      }    }      //Metodo Inorden    private static void inorden(Nodo raiz) {      if (raiz != null) {        inorden(raiz.getNodoIzquierdo());        System.out.print(raiz.getDato()+ " - ");        inorden(raiz.getNodoDerecho());      }    }      //Metodo PostOrden    private static void posOrden(Nodo raiz) {      if (raiz != null) {        posOrden(raiz.getNodoIzquierdo());        posOrden(raiz.getNodoDerecho());        System.out.print(raiz.getDato() + " - ");      }    }    } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | public class Nodo {    //Variables    private int dato;    private Nodo izq;    private Nodo der;      //Constructor    public Nodo(int dato){      this.dato = dato;    }    //Para saber el nodo izquierdo    public Nodo getNodoIzquierdo(){      return izq;    }    //Para saber el nodo derecho    public Nodo getNodoDerecho(){      return der;    }    //Se crea nodo derecho    public void setNodoDerecho(Nodo nodo){      der = nodo;    }    //Se crea nodo Izquierdo    public void setNodoIzquierdo(Nodo nodo){      izq = nodo;    }      public int getDato(){      return dato; |