# TEMA 4.

# Árboles

## CONSIDERACIONES GENERALES.

Se detallan a continuación ejercicios de implementación de funcionalidades de árboles binarios con estructuras dinámicas (referencias). A la hora de abordar la resolución de un ejercicio sobre árboles binarios se recomienda tener en cuenta las siguientes consideraciones generales:

- Identificar si se trata de un caso de solo consulta, es decir, el algoritmo a desarrollar no implica modificar la información inicialmente contenida en el árbol o modificación (inserción, borrado, eliminación).
- Tener en cuenta si el tratamiento permite finalización anticipada. Es decir, si es necesario recorrer toda la estructura (la finalización se produce cuando se cumpla por última vea la condición *arbol==null*) o acaba anticipadamente con el cumplimiento de una condición (por ejemplo encontrar el valor de una clave). En este caso no deberán realizarse posteriores llamadas recursivas.
- Si el algoritmo implica tratamiento de hojas. Recuérdese que la condición que debe cumplirse para que un nodo sea hoja es (*arbol.iz == null) && (arbol.de == null*).
- Si el tratamiento implica consideración de nivel. Como ya se ha indicado se necesita un argumento (*nivel*), con valor inicial = 1 que se incrementa en cada llamada recursiva.
- Cuando se trata de un árbol binario de búsqueda deberá tenerse en cuenta que no se considerará correcto el acceder a una clave mediante exploración exhaustiva del árbol o la no consideración del lugar que ocupan las claves superiores o inferiores a una dada.

**ArbolesCalcularMenorDiferencia**

**Enunciado.**

Dada la siguiente declaración de ARBOL BINARIO:

```
class NodoArbol {
   int clave;
   NodoArbol iz;
   NodoArbol de;
}
public class Arbol {
   NodoArbol raiz;
}
```

**SE PIDE:**

Codificar un método en Java que, recibiendo como parámetro un árbol binario perteneciente al tipo anterior, devuelva la menor diferencia, en valor absoluto, entre la clave contenida en el nodo raíz del árbol y otra clave contenida en algún otro nodo.

**OBSERVACIONES:**

- No se permite la utilización de ninguna estructura de datos auxiliar.
- Al final de la ejecución del método, el árbol deberá permanecer en la misma situación inicial.
- Si el árbol está vacío, el método deberá devolver como resultado el valor cero.
- Si el árbol posee un solo nodo, el método deberá devolver como resultado la clave contenida en ese nodo.

**Orientación.**

El ejercicio (tratamiento principal, *buscarMenorDiferencia*) consiste en realizar sendas llamadas a los subárboles izquierdo y derecho. En cada una de ellas (*menorDiferencia*) se obtiene la menor diferencia entre la raíz y la totalidad de nodos del subárbol correspondiente. El resultado será la menor de las dos. Dicho algoritmo verifica las situaciones excepcionales de árbol vacío, con un solo nodo o con un solo subárbol.

El método recursivo *menorDiferencia* recorre un (sub)árbol y obtiene la menor diferencia entre sus claves y el valor de un argumento que es la clave del árbol inicial. El recorrido puede realizarse en cualquier orden y no existe posibilidad de terminación anticipada. Cada vez que se alcanza la condición de terminación (*arbol == null*) se inicializa el método con la diferencia entre la clave y la raíz. Para ello es necesario pasar como argumento la clave del nodo desde el que se hace la llamada (argumento entero *ant*)[1].

[1] Obsérvese que aunque este cálculo se realiza dos veces no implica múltiple recorrido. Otra posibilidad que no utilizaría este argumento sería inicializar con la constante *Integer.MAX_VALUE*.

**Código.**

```
static int menorDiferencia (NodoArbol arbol, int raiz, int ant) {
   int resul, resulIz;
   if (arbol != null) {
      resulIz = menorDiferencia (arbol.iz, raiz, arbol.clave);
      if (Math.abs (arbol.clave - raiz) < resulIz)
         resulIz = Math.abs (arbol.clave - raiz);
      resul = menorDiferencia (arbol.de, raiz, arbol.clave);
      if (resulIz < resul)
         resul = resulIz;
      }
   else resul = Math.abs (ant - raiz);
   return resul;
}
static int buscarMenorDiferencia (NodoArbol arbol) {
   int resul = 0, resulIz;
   if (arbol != null)
      if (arbol.iz == null)
         if (arbol.de == null)
            resul = arbol.clave;
         else resul = menorDiferencia (arbol.de, arbol.clave, arbol.clave);
      else if (arbol.de == null)
            resul = menorDiferencia (arbol.iz, arbol.clave, arbol.clave);
          else {
            resulIz = menorDiferencia (arbol.iz, arbol.clave, arbol.clave);
            resul = menorDiferencia (arbol.de, arbol.clave, arbol.clave);
            if (resulIz < resul)
               resul = resulIz;
          }
   return resul;
}
static int calcularMenorDiferencia (Arbol arbol) {
   return buscarMenorDiferencia (arbol.raiz);
}
```

**ArbolesComprobarSumaClavesDosNiveles.**

**Enunciado.**

Dada la siguiente declaración de árbol binario:

```
class NodoArbol {
   int clave;
   NodoArbol iz;
   NodoArbol de;
}
public class Arbol {
   NodoArbol raiz;
}
```

**SE PIDE:**

Codificar un algoritmo que, recibiendo un árbol perteneciente al tipo anteriormente descrito y dos números enteros, *n1* y *n2,* indique si la suma de las claves que se encuentra a nivel *n1* es igual a la suma de las claves que se encuentra a nivel *n2*.

**OBSERVACIONES:**

- Sólo se permitirá la realización de un único recorrido en el árbol.
- No se permitirá la utilización de ninguna estructura de datos auxiliares.
- Se supone que en el árbol siempre van a existir claves en los niveles *n1* y *n2*.

**Orientación.**

Los argumentos iniciales son el *arbol*, y los niveles *n1* y *n2*. Los tres pueden pasarse por valor pues no se va a realizar ninguna modificación. Dado que se trata de realizar un recorrido por el árbol en el que hay que tener constancia del *nivel* del nodo visitado en cada momento será necesario incluirlo como argumento adicional. Se deberá pasar por valor para que cada instancia almacene el suyo propio. El valor inicial, proporcionado por un módulo externo al módulo recursivo, tendrá el valor de 1 (nivel de la raíz), y las llamadas recursivas se lanzarán con *nivel+1*.

El algoritmo tiene que recorrer el *arbol* completamente (no hay terminación anticipada) verificando si se está accediendo a un nodo de *nivel n1* o *n2*. Dado que solo es necesario conocer la diferencia entre dos resultados se puede utilizarse un método entero recursivo, *suma*, que actuando como acumulador sume, por ejemplo, las claves de los nodos de nivel *n1* y reste las que se encuentre a nivel *n2*. Si el resultado final del método recursivo *suma* es 0 el método de llamada devolverá *true* y *false* en caso contrario.

**Código.**

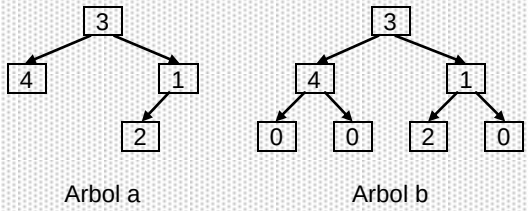
```
static int suma (NodoArbol arbol, int n1, int n2, int nivel) {
    int aux = 0;
    if (arbol != null) {
        if (nivel == n1)
            aux = arbol.clave;
        else if (nivel == n2)
                aux = -arbol.clave;
             else aux = 0;
        if ((nivel < n2) || (nivel < n1))
            aux = aux + suma (arbol.iz, n1, n2, nivel+1) + suma (arbol.de, n1, n2, nivel+1);
    }
    return aux;
}
static boolean comprobarSumaClavesDosNiveles (Arbol arbol, int n1, int n2) {
    boolean aux = false;
    if (arbol.raiz != null)
        aux = (suma (arbol.raiz, n1, n2, 1) == 0);
    return aux;
}
```

**ArbolesTransformarEnCompleto**

**Enunciado.**

Dada la siguiente declaración de árbol binario

```
public class NodoArbol {
   int clave;
   NodoArbol iz;
   NodoArbol de;
   NodoArbol () {
      clave = 0;
      iz = null;
      de = null;
   }
}
public class Arbol {
   NodoArbol raiz;
}
```




Arbol a Arbol b

**SE PIDE:** Codificar un método en Java que, recibiendo como parámetro un árbol binario perteneciente al tipo anterior y un número entero n que representa la altura del árbol, transforme dicho árbol en otro que sea completo, incluyendo en él una serie de nodos adicionales.

**OBSERVACIONES:**

- Sólo se permite la realización de un único recorrido en el árbol.
- No se permite la utilización de ninguna estructura de datos auxiliar.
- Los nuevos nodos que se incluyan en el árbol, deberán contener una clave igual a 0.
- Un árbol de altura n se dice que es completo si, y sólo si, el número de nodos de cada nivel i >= 1 es igual a $2^{i+1}$. Es decir, en el nivel 1 (la raíz) deberá haber 1 nodo, en el nivel 2 deberá haber 2 nodos, en el nivel 3 deberá haber 4 nodos, en el nivel 4 deberá haber 8 nodos, y así sucesivamente.

**EJEMPLO:**

Dado el árbol (a) mostrado en la figura, el método deberá devolver el árbol (b) mostrado en dicha figura.

**Orientación.**

Se trata de hacer un recorrido completo del *arbol* teniendo constancia del nivel. Se necesita un argumento (*nivel*), pasado por valor e inicializado a 1 que se incrementa en una unidad en cada llamada.

Cada vez que se alcance una condición de terminación (*nodo==null*) se verifica si el nivel de la hoja correspondiente (valor actual de *nivel* –1) es inferior a la altura del *arbol* (*n*). En caso de ser así se procede a un tratamiento **recursivo** de inserción hasta que aparezcan todas las hojas a la misma altura (la del árbol).

**Código.**

```
    static NodoArbol transformarEnCompleto(NodoArbol nodo, int n, int nivel) {
        if (nodo != null) {
                nodo.iz = transformarEnCompleto (nodo.iz, n, nivel+1);
                nodo.de = transformarEnCompleto (nodo.de, n, nivel+1);
        }
        else {
            if (nivel<=n) {
                nodo = new NodoArbol(0);
                if (nivel<n) {
                    nodo.iz = transformarEnCompleto (nodo.iz, n, nivel+1);
                    nodo.de = transformarEnCompleto (nodo.de,n,nivel+1);
                }
            }
        }
        return nodo;
    }
    static void transformarEnCompleto (Arbol arbol, int n) {
        arbol.raiz = transformarEnCompleto (arbol.raiz, n, 1);
}
```

**ArbolesEliminarHojasDebajoDeUnNivel**

**Enunciado.**

Dada la siguiente declaración de Árbol binario:

```
public class NodoArbol {
   int clave;
   NodoArbol iz;
   NodoArbol de;
}
public class Arbol {
   NodoArbol raiz;
}
```

**SE PIDE:**

Codificar un método que elimine todas las hojas cuyo nivel sea mayor o igual a un valor *n* que se pasa como parámetro.

**NOTAS:**

- No está permitido posicionarse o acceder más de una única vez sobre cada nodo del árbol.
- Considérese que la raíz del árbol se encuentra a nivel 1.

**Orientación.**

Es necesario realizar un recorrido completo del árbol, necesariamente en preorden.

Puesto que se trata de un proceso en que hay que tener constancia del nivel en que se encuentra será necesario utilizar un argumento (*nivel*, pasado por valor e inicializado a 1 desde el módulo de llamada) que lo indique. Cada llamada recursiva deberá incrementar en una unidad el valor de este argumento.

Se verificará el cumplimiento de que el nivel del nodo visitado es igual o superior a *n* y que se verifica la condición de hoja (*nodo.iz == null) && (nodo.de == null).*

**Código.**

```
static NodoArbol eliminarHojasDebajoDeUnNivel (NodoArbol nodoArbol,int  n,int nivel) {
   NodoArbol resul = nodoArbol;
   if (nodoArbol != null) {
      if ((nodoArbol.iz == null) && (nodoArbol.de == null) && (nivel>=n))
         resul = null;
      else{
         nodoArbol.iz = eliminarHojasDebajoDeUnNivel  (nodoArbol.iz, n, nivel+1);
         nodoArbol.de = eliminarHojasDebajoDeUnNivel  (nodoArbol.de, n, nivel+1);
      }
   }
   return resul;
}
static void eliminarHojasDebajoDeUnNivel (Arbol arbol, int n) {
   arbol.raiz = eliminarHojasDebajoDeUnNivel (arbol.raiz, n, 1);
}
```

**ArbolesBusquedaReorganizable**.

**Enunciado.**

Dada la siguiente declaración de árbol binario:

```
class NodoArbol {
   int clave;
   NodoArbol iz;
   NodoArbol de;
}
public class Arbol {
   NodoArbol raiz;
}
```

Se sabe que, una vez que se efectúa la búsqueda de una clave x en dicho árbol, las siguientes N operaciones que se produzcan en el árbol van a ser también búsquedas de esa misma clave x (N es un número muy grande).

**SE PIDE:**

Codificar un método en Java que, recibiendo como parámetros un árbol binario perteneciente al tipo anterior y una clave entera *x*, busque la clave *x* en el árbol, devolviendo *true* en caso de que la clave *x* se encuentre en el árbol, y *false* en caso contrario. El método deberá minimizar el número de accesos al árbol durante todas las operaciones de búsqueda de la clave *x* que se produzcan.

**OBSERVACIONES:**

- No se permite la utilización de ninguna estructura de datos auxiliar.
- Sólo se permite la realización de un único recorrido en el árbol.
- Se supone que en el árbol no existen claves repetidas.
- Si se considera oportuno, se podrán reorganizar algunas claves del árbol.

**Orientación**

Se trata de encontrar la clave (*x*) en el árbol y, si existe y no está actualmente en la raíz del mismo, colocarla en la raíz para minimizar los accesos posteriores. Existen dos posibilidades:

- Una vez encontrada la clave, en ese nodo ponemos el valor de la raíz anterior, y a la vuelta de la recursividad, ponemos en la raíz el valor buscado. Para esto, necesitaremos llevarnos en las sucesivas llamadas recursivas el valor de la raíz.
- Cambiar la ubicación de alguna(s) clave(s) del árbol: vamos arrastrando el valor de la clave anterior (*ant*), y si hemos encontrado la clave buscada, vamos poniendo en cada nodo la clave que tenía su padre, hasta llegar a la raíz, donde pondríamos la clave buscada. Esto exigiría realizar dos tratamientos de cada clave:
  - Comprobamos si es la clave buscada, en caso de serlo, lo indicaríamos en una variable booleana local *esta*
  - Si no *esta* buscamos por su hijo izquierdo,
  - Si no lo hemos encontrado en el hijo izquierdo ni *esta*, buscamos por la derecha,
  - Si hemos encontrado la clave, pondríamos en la clave actual el valor de la clave del padre
  - Por último, al llegar a la raíz, pondremos como clave (como en el caso anterior) la que nos habían pasado inicialmente como argumento.

**Código.**

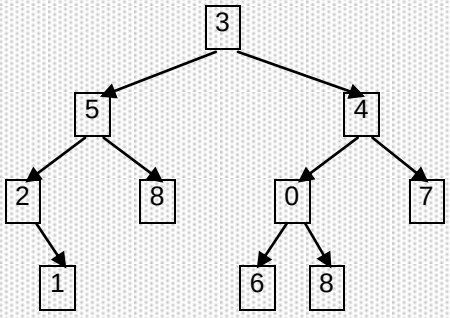
```
static boolean busqueda (NodoArbol arbol, int x, int ant) {
    boolean esta;
    int actual;
    if (arbol != null) {
        actual = arbol.clave;
        if (actual == x) {
            esta = true;
            arbol.clave = ant;
        }
        else {
            esta = busqueda (arbol.iz,x,actual);
            if (!esta)
                esta = busqueda (arbol.de,x,actual);
            if (esta)
                arbol.clave = ant;
        }
    }
    else esta = false;
    return esta;
}
static boolean busquedaReorganizable (Arbol arbol, int x) {
    return busqueda (arbol.raiz, x, x);
}
```

**ArbolesComprobarSiAlturaIgualAnchura.**

**Enunciado.**

Dada la siguiente declaración de árbol binario

```
class NodoArbol {
   int clave;
   NodoArbol iz;
   NodoArbol de;
}
public class Arbol {
   NodoArbol raiz;
}
```




Dado el *TAD Pila de NodoArbol* con las siguientes operaciones:

```
void inicializarPila ();
boolean pilaVacia ();
void apilar (NodoArbol x);
NodoArbol desapilar () throws Pilavacia;
```

Y dado el *TAD Cola de NodoArbol* con las siguientes operaciones:

```
void inicializarCola ();
boolean colaVacia ();
void encolar (NodoArbol elem);
NodoArbol desencolar () throws Colavacia;
```

**SE PIDE:**

Codificar un método en Java que, recibiendo como parámetro un árbol binario perteneciente al tipo anterior, determine si la anchura de dicho árbol es igual a su altura.

**OBSERVACIONES:**

- Sólo se permite la realización de un único recorrido en el árbol.
- No se permite la utilización de ninguna estructura de datos auxiliar, a excepción de una pila perteneciente al TAD Pila anterior, o una cola perteneciente al TAD Cola anterior.
- Se define altura de un árbol como el nivel máximo al que se encuentre alguno de los nodos del árbol. Asimismo se define anchura de un árbol como el número de nodos que se encuentren en el nivel con mayor número de nodos en el árbol.
- Al finalizar el proceso, el árbol deberá quedar en la misma situación inicial.

**EJEMPLO:**

Dado el árbol mostrado en la figura, tanto su altura como su anchura valen 4. Por tanto, el método deberá determinar que su altura es igual a su anchura.

**Orientación.**

Se trata de una ligera variante del algoritmo básico de recorrido en amplitud con conocimiento del nivel explicado en teoría (y por tanto, utilizaremos como estructura de datos auxiliar una cola).

Dicha variante consiste en añadir una nueva variable, *anchura* que se actualice con el mayor de los valores alcanzados en cada nivel (variable *actual*).

El valor devuelto por el método será el resultado de comparar las variables *altura* y *anchura.*

**Código.**

```
static boolean comprobarSiAlturaIgualAnchura (Arbol arbol) {
    NodoArbol p = arbol.raiz;
    Cola c = new TadCola ();
    int actual, siguiente = 1, contador, altura = 0 , anchura = 0;
    if (p != null)
    c.encolar (p);
    while (!c.colaVacia()) {
        actual = siguiente;
        siguiente = 0;
        contador = 0;
        altura++;
        if (anchura < actual)
            anchura = actual;
        while (contador<actual) {
            p = c.desencolar ();
            contador++;
            if (p.iz != null) {
                c.encolar (p.iz);
                siguiente++;
            }
            if (p.de != null) {
                c.encolar (p.de);
                siguiente++;
            }
        }
    }
    return (anchura == altura);
}
```

**ArbolesMostrarClavesHojasPosicionPar.**

**Enunciado.**

Dada la siguiente declaración de Árbol Binario de números enteros**:**

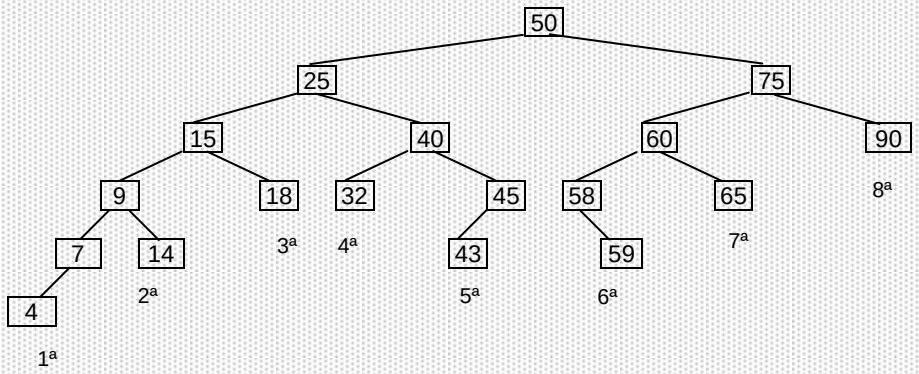
```
public class NodoArbol {
   int clave;
   NodoArbol iz;
   NodoArbol de;
}
public class Arbol {
   NodoArbol raiz;
}
```

**SE PIDE:**

Implementar un método en Java, que recibiendo como parámetro un *árbol* perteneciente al tipo anterior, muestre, en orden topológico, las claves de las hojas que ocupan posiciones pares.

**EJEMPLO:**

Dado el *árbol* de la figura siguiente:




el método mostrará la secuencia de valores: 14, 32, 59, 90

**OBSERVACIONES:**

- No se permite la utilización de ninguna estructura de datos auxiliar.
- Sólo se permite la realización de un único recorrido del *árbol*.
- Al final de la ejecución del método, el *árbol* deberá permanecer con la estructura y el contenido iniciales

**Orientación.**

En este ejercicio hay que realizar un recorrido en orden central, contando el número de hojas. Se utiliza la variable *numHojas* de la clase *Mostrar* para ir guardando el número de hojas que hemos encontrado hasta el momento. Cada vez que localicemos una hoja (*arbol.iz == null && arbol.de == null*), incrementaremos *numHojas*. Si *numHojas* es par (*numHojas % 2 == 0*), escribiremos la clave.

**Código.**

```
static class Mostrar {
    static int numHojas=0;
    static void muestraHojasPares (NodoArbol arbol) {
        if (arbol != null)
            if ((arbol.iz == null && arbol.de == null)) {
                numHojas++;
                if (numHojas % 2 == 0)
                    System.out.println(arbol.clave);
            }
            else {
                muestraHojasPares (arbol.iz);
                muestraHojasPares (arbol.de);
            }
    }
}
static void muestraHojasPares (Arbol arbol) {
    Mostrar.muestraHojasPares (arbol.raiz);
}
```

**ArbolesBusquedaComprobarSiClaveEnHoja**.

**Enunciado.**

Dada la siguiente declaración de Árbol Binario de BÚSQUEDA:

```
public class NodoArbol {
   int clave;
   NodoArbol iz;
   NodoArbol de;
}
public class Arbol {
   NodoArbol raiz;
}
```

**SE PIDE:**

Codificar un método en Java que indique si una determinada clave, que se recibe como parámetro, se encuentra o no en una hoja.

**Orientación.**

El ejercicio consiste en recorrer el árbol buscando la clave que coincida con el dato suministrado. Dado que se trata de un árbol binario de búsqueda, ésta es dirigida: se realizan llamadas recursivas por la derecha o por la izquierda hasta encontrarla.

Se reconoce que no existe la clave buscada por haber llegado a algún extremo del árbol (*arbol ==null*) lo que proporcionará un resultado *false*.

En caso de encontrar la clave durante el recorrido pueden suceder dos casos (y ya no habrá posteriores llamadas recursivas):

- Se ha encontrado en una hoja (*arbol.iz==null && arbol.de==null*). El resultado es *true*.
- Se ha encontrado en un nodo que no es hoja. El resultado es *false*.

**Código.**

```
static boolean comprobarSiHoja (NodoArbol arbol, int dato) {
   boolean resul;
   if (arbol != null)
      if (arbol.clave == dato)
         if ((arbol.iz == null) && (arbol.de == null))
            resul = true;
         else resul = false;
      else if (arbol.clave > dato)
              resul = comprobarSiHoja (arbol.iz,dato);
           else resul = comprobarSiHoja (arbol.de,dato);
                else resul = false;
   return resul;
}
static boolean comprobarSiClaveEnHoja (Arbol arbol, int dato) {
   return comprobarSiHoja (arbol.raiz, dato);
}
```

**ArbolesBusquedaBuscarMenorDeLasClavesMayores**.

**Enunciado.**

Dada la siguiente declaración de Árbol Binario de BÚSQUEDA:

```
public class NodoArbol {
   int clave;
   NodoArbol iz;
   NodoArbol de;
}
public class Arbol {
   NodoArbol raiz;
}
```

**SE PIDE:**

Codificar un método en Java que devuelva una referencia que señale al nodo que contiene la menor de las claves mayores, de entre sus descendientes, a una que se pasa como parámetro. En el caso de no existir ninguna clave mayor, el método deberá devolver *null*.

**NOTA:**

No se permitirá la utilización de estructuras de datos auxiliares.

**Orientación.**

Dada la naturaleza del árbol binario de búsqueda, la menor de las claves mayores se encontraría, en su caso, en el extremo izquierdo del subárbol derecho del nodo que contiene el *dato* proporcionado como argumento. Para que devuelva un valor válido deberá existir la clave buscada así como un subárbol derecho no vacío a partir de dicha clave en caso contrario.

El proceso se ejecuta en dos etapas, ambas de naturaleza recursiva.

- La primera consiste en realizar la búsqueda del *dato* recibido como argumento, una vez localizado se verifica la existencia de un subárbol derecho no nulo (*arbol.de* != *null*). En caso afirmativo se realiza la llamada inicial al segundo método (*laMenor*) pasándole como argumento inicial la referencia a dicho subárbol.
- El método *laMenor* recorre el subarbol derecho de la clave buscada, buscando un nodo sin hijo izquierdo, que será el resultado devuelto.

**Código.**

```
static NodoArbol laMenor (NodoArbol arbol) {
    NodoArbol resul;
    if (arbol.iz != null)
        resul = laMenor (arbol.iz);
    else resul = arbol;
    return resul;
}
static NodoArbol buscarMenor (NodoArbol arbol, int dato) {
    NodoArbol resul;
    if (arbol != null)
        if (arbol.clave < dato)
            resul = buscarMenor (arbol.de, dato);
        else if (arbol.clave > dato)
                    resul = buscarMenor (arbol.iz,dato);
                else if (arbol.de == null)
                            resul = null;
                        else resul = laMenor (arbol.de);
    else resul = null;
    return resul;
}
static NodoArbol buscarMenorDeLasClavesMayores (Arbol arbol, int dato) {
    return buscarMenor (arbol.raiz, dato);
}
```

**ArbolesBusquedaVerificarSiAntecesoresAscendentes**.

**Enunciado.**

Dada la siguiente declaración de ÁRBOL BINARIO DE BÚSQUEDA:

```
public class NodoArbol {
   int clave;
   NodoArbol iz;
   NodoArbol de;
}
public class Arbol {
   NodoArbol raiz;
}
```

**SE PIDE:**

Codificar un método booleano en Java que, recibiendo como parámetro un árbol binario de búsqueda perteneciente al tipo anterior y una clave entera x, determine si los antecesores de x, empezando desde el nodo raíz, constituyen una secuencia de claves ascendente.

**OBSERVACIONES:**

- No se permite la utilización de ninguna estructura de datos auxiliar. El algoritmo deberá minimizar el número de nodos visitados.
- Si el árbol está vacío, el método deberá devolver el valor *false*.
- Si la clave x no se encuentra en el árbol, el método deberá devolver así mismo el valor *false*.
- Si la clave x no posee antecesores, el método deberá devolver el valor *true*.

**Orientación.**

Dada la naturaleza del árbol binario de búsqueda, la secuencia ascendente de claves antecesoras se encontrará en el camino derecho del mismo. En consecuencia, en caso de existir la clave buscada podrá bien formar parte del camino derecho o bien **ser el hijo izquierdo** de su ascendiente inmediato (padre)[2].

El algoritmo recursivo (la llamada se realiza solo desde el hijo derecho: *arbol.de*) presenta las siguientes condiciones de terminación:

- Pesimista: Se alcanza el final del camino derecho (*arbol == null*). Significa que no se ha encontrado la clave buscada y el método devuelve *false*.
- Anticipada:
  - Se encuentra la clave en el camino derecho (o nodo raíz en la llamada inicial), por tanto sus antecesores constituyen una clave ascendente (o la clave buscada es la raíz) y el resultado es *true*.
  - La clave buscada es menor que la del nodo actual (se abandona el camino derecho). Pueden darse las siguientes circunstancias:
    - Existe un hijo izquierdo de dicho nodo cuya clave coincide con el dato buscado. El resultado es *true*.
    - En cualquier otra circunstancia o bien no existe la clave buscada o bien sus antecesores no constituyen una secuencia ascendente por lo que el método devolverá el valor *false*.

---

[2] Se trata de una orientación que evita pasar como argumento la clave del nodo antecesor para verificar si la secuencia es ascendente. Se sugiere al alumno la realización de este ejercicio realizando un recorrido dirigido pasando como argumento la clave del antecesor inmediato y produciendo terminación anticipada si en el camino de búsqueda de la clave (antes de encontrarla) se encuentra un valor anterior superior al actual (resultado *false*) o en el momento de encontrar con éxito la clave (resultado *true*). La terminación pesimista (*arbol == null*) se produciría si la clave buscada no se encuentra en el *arbol*.

Obsérvese que el algoritmo recursivo contempla como caso particular las situaciones de excepción expresadas en el enunciado.

**Código.**

```
static boolean verificarAntecesoresAscendentes (NodoArbol arbol, int dato) {
   boolean resul;
   if (arbol != null)
      if (arbol.clave > dato)
         if (arbol.iz == null)
            resul = false;
         else if (arbol.iz.clave == dato)
                 resul = true;
              else resul = false;
      else if (arbol.clave < dato)
              resul = verificarAntecesoresAscendentes (arbol.de, dato);
           else resul = true;
   else resul = false;
   return resul;
}
static boolean verificarSiAntecesoresAscendentes (Arbol arbol, int dato) {
   return verificarAntecesoresAscendentes (arbol.raiz, dato);
}
```

**ArbolesTadArbolBinarioCalcularMediana**.

**Enunciado.**

Dado el TAD Árbol Binario de Números Enteros Positivos con las siguientes operaciones:

```
public interface Arbol {
   void crearNodo ();
     /*Crea un nuevo nodo en el árbol, que queda apuntando a ese nodo*/
   int obtenerClave ();
     /*Devuelve la clave contenida en el nodo del árbol apuntado por Puntero*/
   NodoArbol devolverRaiz ();
     /*Devuelve una referencia a la raiz del arbol*/
   NodoArbol devolverHijoIzquierdo ();
     /*Devuelve el hijo izquierdo del árbol*/
   NodoArbol devolverHijoDerecho ();
     /*Devuelve el hijo derecho del árbol*/
   void ponerClave (int Clave);
     /*pone la clave pasada como argumento en la raíz del ábol*/
   void ponerReferencia (NodoArbol nuevoArbol);
     /*Hace que el árbol apunte al mismo sitio que nuevoArbol*/
   void ponerHijoIzquierdo (NodoArbol arbolAsignado);
     /*Hace que el hijo izquierdo del arbol apunte ahora a arbolAsignado*/
   void ponerHijoDerecho (NodoArbol arbolAsignado);
     /*Hace que el hijo derecho del arbol, apunte ahora a arbolAsignado*/
   void asignarNulo ();
     /*Hace que el arbol tome el valor null*/
   boolean esNulo ();
     /*Devuelve true si el arbol tiene valor null y false en caso contrario*/
   boolean iguales (NodoArbol otroArbol);
     /*Devuelve true si el árbol y otroArbol apuntan al mismo sitio, false en caso
     contrario*/
}
```

**SE PIDE:**

Codificar un método booleano en Java que, recibiendo como parámetro un Árbol Binario de Búsqueda perteneciente al TAD anterior, determine si la ***mediana*** de las claves contenidas en dicho árbol se encuentra en el nodo raíz.

**OBSERVACIONES:**

- No se permite la utilización de estructuras de datos auxiliares.
- Solo se permite la realización de un único recorrido en el árbol.
- Se supone que el número de claves del árbol es siempre impar.
- Se denomina ***mediana*** de un conjunto de claves a aquella clave *m* tal que el número de claves menores que *m* es igual al número de claves mayores que *m*.

**Orientación.**

Dado que se trata de un árbol **de búsqueda** todas las claves situadas a la izquierda del nodo raíz son inferiores al valor de ella y todas las situadas a la derecha superiores. Por tanto, si el número de nodos de los subárboles izquierdo y derecho coinciden el método devolverá *true* (la mediana está contenida en el nodo raíz) y *false* en caso contrario.

El método solicitado implica realizar (si el árbol no llega vacío) sendas llamadas a un método (*contar*) que cuenta los nodos de los subárboles izquierdo y derecho.

La adaptación del ejercicio al TAD suministrado supone utilizar únicamente las operaciones del TAD de forma que se realizan las siguientes sustituciones:

- *arbol == null* por *arbol.esNulo()*.
- *arbol.iz (acceso)* por *arbol.devolverHijoIzquierdo ()*
- *arbol.de (acceso)* por *arbol.devolverHijoDerecho ()*
- *arbol = nuevoNodoArbol* por *arbol.ponerReferencia (nuevoNodoArbol)*

**Código.**

```
static int contar (TadArbol arbol) {
   TadArbol auxIz= new TadArbol (), auxDe= new TadArbol ();
   NodoArbol auxNodoIz, auxNodoDe;
   int resul;
   if (!arbol.esNulo ()) {
      auxNodoIz = arbol.devolverHijoIzquierdo () ;
      auxIz.ponerReferencia (auxNodoIz);
      auxNodoDe = arbol.devolverHijoDerecho () ;
      auxDe.ponerReferencia (auxNodoDe);
      resul = 1 + contar (auxIz) + contar (auxDe);
   }
   else resul = 0;
   return resul;
}

static boolean medianaEnRaiz (TadArbol arbol) {
   boolean resul;
   TadArbol auxIz= new TadArbol (), auxDe= new TadArbol ();
   NodoArbol auxNodoIz, auxNodoDe;
   if (arbol.esNulo ())
      resul = true;
   else {
      auxNodoIz = arbol.devolverHijoIzquierdo () ;
      auxIz.ponerReferencia (auxNodoIz);
      auxNodoDe = arbol.devolverHijoDerecho () ;
      auxDe.ponerReferencia (auxNodoDe);
      resul = (contar (auxIz) == contar (auxDe));
   }
   return resul;
}
```