TP - Unidad 05-C
Árboles De Búsqueda. Caso de Uso: Soporte para Índices.

Ejercicio 1

Mostrar gráficamente, paso a paso, cómo quedaría la inserción en un BST si los datos se insertan en el siguiente orden: 50 60 80 20 70 40 44 10 40

Ejercicio 2

Generar un proyecto mvn con las siguientes interfaces (están en Campus), para el BST y para el nodo interno respectivamente:

```
package core;

public interface BSTreeInterface<T extends Comparable<? super T>> {

void insert(T myData);

void preOrder();

void postOrder();

void inOrder();

NodeTreeInterface<T> getRoot();

int getHeight();

}
```

```
package core;

public interface NodeTreeInterface<T extends Comparable<? super T>> {

T getData();

NodeTreeInterface<T> getLeft();

NodeTreeInterface<T> getRight();

}
```

2.1) Implementar el método insert dos versiones: la que resuelve todo desde BST y la que delega la inserción en la clase Node.

2.2) Implementar todos los demás métodos.

Caso de Uso:

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

BST<Integer> myTree = new BST<>();
myTree.inOrder();
myTree.preOrder();
myTree.postOrder();

// y este?
myTree.insert(50);
myTree.insert(60);
myTree.insert(80);
myTree.insert(20);
myTree.insert(70);
myTree.insert(40);
myTree.insert(44);
myTree.insert(10);
myTree.insert(40);

myTree.inOrder();
myTree.preOrder();
myTree.postOrder();
```

Ejercicio 3

Vamos a realizar la versión gráfica del mismo BST anterior.

3.1) Chequear que todos los métodos están bien implementados.
Tip: para que funcione, se debe haber implementado correctamente los métodos:

```
@Override
public NodeTreeInterface<T> getRoot() {
	…
}


@Override
public int getHeight() {
	…
}
```

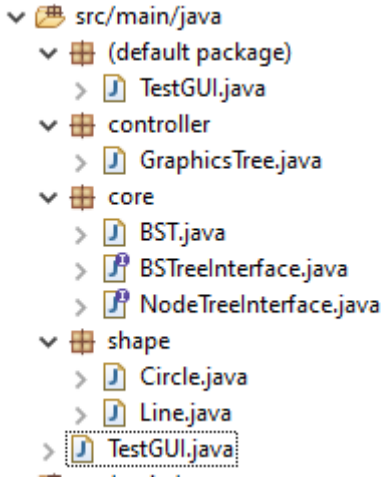
3.2) La versión GUI usa javafx que ya no es parte de Java. Agregan al pom.xml las siguientes dependencias:

```
<dependencies>
<dependency>
      <groupId>org.openjfx</groupId>
      <artifactId>javafx-fxml</artifactId>
      <version>11</version>
</dependency>
<dependency>
  <groupId>org.openjfx</groupId>
      <artifactId>javafx-base</artifactId>
      <version>11</version>
</dependency>
<dependency>
      <groupId>org.openjfx</groupId>
      <artifactId>javafx-controls</artifactId>
      <version>11</version>
  </dependency>
  <dependency>
      <groupId>org.openjfx</groupId>
      <artifactId>javafx-graphics</artifactId>
      <version>11</version>
  </dependency>
</dependencies>
```

3.3) Las 2 interfaces están dentro del package core. Ahora agregar al proyecto (Bajar de Campus):

- GraphicsTree.java (en package controller)
- Circle.java y Line.java (en package shape)
- TestGUI.java (podría estar en package default). Es la aplicación de ejemplo javafx.

Tenemos entonces la siguiente estructura de clases:



3.4) Bajar desde https://gluonhq.com/products/javafx/

Las bibliotecas nativas V 11.0.2 (para Win, Linux o Mac): JavaFX Windows/Linux/Mac SDK

Descompactar en un directorio con permisos.

Ejemplo:

C:\Users\lgomez\Downloads\javafx-sdk-11.0.2

O para Linux/Mac:

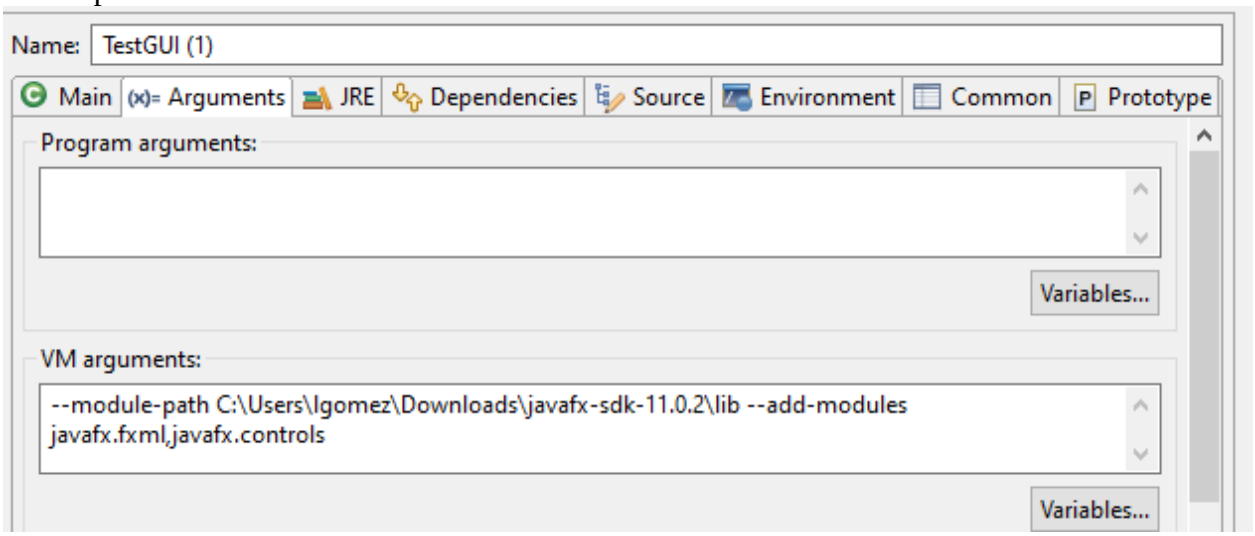
/tmp/javafx-sdk-11.0.2

Estos directorios contienen dentro un directorio lib (y en Win directorio también bin)

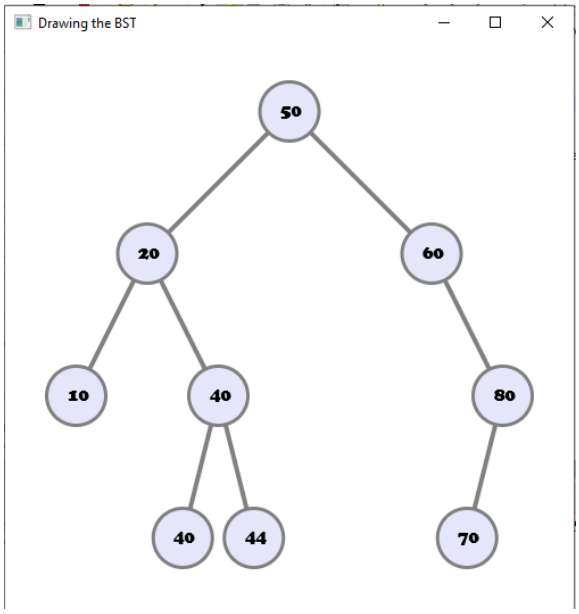
3.5) Para ejecutar desde eclipse/IntelliJ hay que setear parámetros el Java VM. La aplicación es la TestGUI, los parámetros deben indicar el path completo hasta el directorio lib donde se descompactó la biblioteca nativa del punto 3.4
Ejemplo:

--module-path C:\Users\lgomez\Downloads\javafx-sdk-11.0.2\lib --add-modules javafx.fxml,javafx.controls

En eclipse:



Al ejecutar se debe obtener, la versión gráfica del BST anterior.



Ejercicio 4

El método **createModel()** nos permite cambiar el contenido del BSTree.

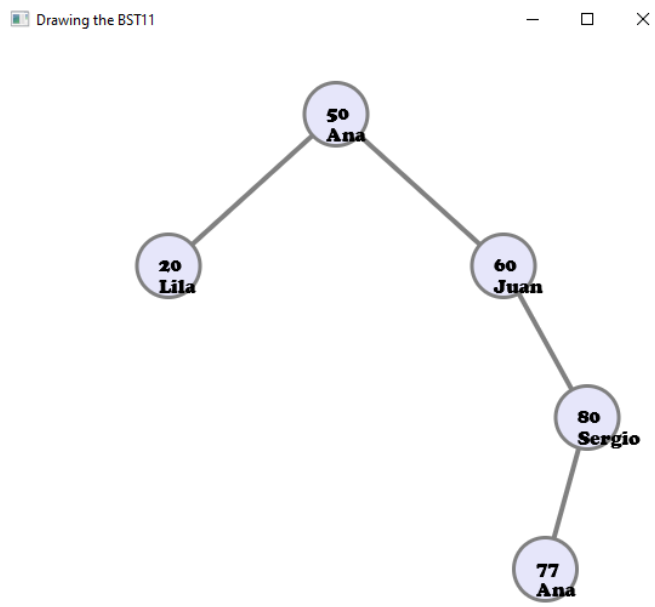
4.1) Crear un TAD opaco **Person** que contenga como mínimo legajo y nombre. El mismo debe participar del BST ordenando por legajo.

4.2) Crear varias instancias de Person, agregarlas a la colección y probar su graficación en el árbol.

Caso de Uso

```
…
private BSTree<Person> createModel() {
   BSTree<Person> myTree = new BSTree<>();
   myTree.insert(new Person(50, "Ana"));
   myTree.insert(new Person(60, "Juan"));
   myTree.insert(new Person(80, "Sergio"));
   myTree.insert(new Person(20, "Lila"));
   myTree.insert(new Person(77, "Ana"));
   return myTree;
}
```

Al ejecutar el TestGUI con este nuevo modelo se debería obtener:



Ejercicio 5

Agregar en la interface BSTreeInterface los siguiente métodos e implementarlos en BST

5.1)
boolean contains(T myData);

El método no debe recorrer innecesariamente el árbol. Nunca debe lanzar excepción.

5.2)
T getMax();
T getMin();

Los métodos no debe recorrer innecesariamente el árbol.
Si el árbol no tiene elementos devuelve null.

5.3)
void printByLevels()

El mismo debe imprimir la información del árbol de la siguiente forma: primero la info del nodo en nivel 0 (la raíz), luego la info de nodos en nivel 1 (hijos), luego nivel 2, y así sucesivamente.

Caso de Uso:

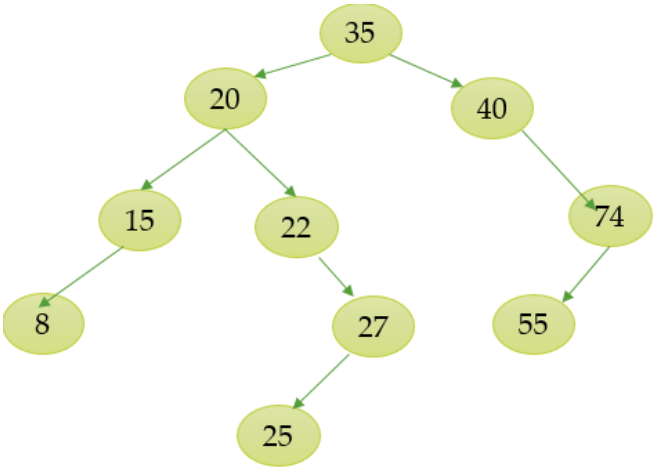
```
BST<Integer> myTree= new BST<>();
myTree.insert(35);
myTree.insert(74);
myTree.insert(20);
myTree.insert(22);
myTree.insert(55);
```

myTree.insert(15);
myTree.insert(8);
myTree.insert(27);
myTree.insert(25);
myTree.printByLevels();

debería obtenerse 35 20 74 15 22 55 8 27 25

Ejercicio 6

6.1) A partir del siguiente BST, mostrar gráficamente cómo queda paso a paso el BST luego de aplicar las siguientes operaciones de borrado y qué reglas se usaron. Usar predecesor inorder: 40, 35 y 8 (en ese orden)



6.2) Agregar a la interface el método `void delete(T myData);` e implementarlo

Ejercicio 7

7.1) Se quiere que BSTree ofrezca la posibilidad de ser iterado, esto es, que implemente la interface Iterable<T>.
Cambiar la interface BSTreeInterface para que extienda a Iterable.

Implementar el método y testearlo

```
@Override
public Iterator<T> iterator() {
	…
}
```

El iterador devolverá los **elementos por niveles**, como lo hicimos previamente.

Caso de Uso:

```
BST<Integer> myTree = new BST<>();
myTree.insert(35);
myTree.insert(74);
myTree.insert(20);
myTree.insert(22);
myTree.insert(55);
myTree.insert(15);
myTree.insert(8);
myTree.insert(27);
myTree.insert(25);
for (Integer data : myTree) {
   System.out.print(data +  " ");
}
// Puedo hacerlo múltiples veces…
System.out.println("\n\nUna vez más…\n");
myTree.forEach( t-> System.out.print(t + " ") );
```

debería obtenerse
35 20 74 15 22 55 8 27 25
Una vez más
35 20 74 15 22 55 8 27 25

7.2) La siguiente es la versión iterativa del método inOrder:

```
// iterativa
public void inOrderIter() {
  Stack<NodeTreeInterface<T>> stack=  new Stack<>();

  NodeTreeInterface<T> current = root;
  while ( ! stack.isEmpty() || current != null) {
  if (current != null) {
        stack.push(current);
        current= current.getLeft();
  }
  else {
        NodeTreeInterface<T> elementToProcess = stack.pop();
        System.out.print(elementToProcess.getData() + "\t");
        current= elementToProcess.getRight();
        }
  }
 }
```

Cambiar para que el iterador devuelva los **elementos inOrder**, como lo hicimos previamente.

```
@Override
public Iterator<T> iterator() {
```

```
        …
}
```

Para ello, tomando la idea de `inOrderIter() escribir la versión Iterator de inOrder()`

Caso de Uso:

```
BST<Integer> myTree = new BST<>();
myTree.insert(35);
myTree.insert(74);
myTree.insert(20);
myTree.insert(22);
myTree.insert(55);
myTree.insert(15);
myTree.insert(8);
myTree.insert(27);
myTree.insert(25);
for (Integer data : myTree) {
   System.out.print(data +  " ");
}
// Puedo hacerlo múltiples veces…
System.out.println("\n\nUna vez más…\n");
myTree.forEach( t-> System.out.print(t + " ") );
```

debería obtenerse

```
8 15 20 22 25 27 35 55 74
```

Una vez más

```
8 15 20 22 25 27 35 55 74
```

7.3) Hasta ahora tenemos 2 formas de iterador. Se quiere que se el usuario pueda setear el iterador deseado, a medida que no necesita. De no setear ninguno se asume impresión por niveles. Agregar a la interface

```
public interface BSTreeInterface<T extends Comparable<? super T>> extends Iterable<T> {
 enum Traversal { BYLEVELS, INORDER }

 void setTraversal(Traversal traversal);
 …
}
```

E implementarlo en la clase BST

Caso de Uso:

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

BST<Integer>  myTree= new BST<>();
myTree.insert(35);   myTree.insert(74);
myTree.insert(20);   myTree.insert(22);
myTree.insert(55);   myTree.insert(15);
myTree.insert(8);    myTree.insert(27);
myTree.insert(25);
System.out.println("\n\nDefault Traversal…\n");
myTree.forEach( t-> System.out.print(t + " ") );
myTree.setTraversal(Traversal.INORDER);
System.out.println("\n\nUna vez más INORDER\n");
myTree.forEach( t-> System.out.print(t + " ") );
myTree.setTraversal(Traversal.BYLEVELS);
System.out.println("\n\nUna vez más BYLEVELS\n");
myTree.forEach( t-> System.out.print(t + " ") );
myTree.setTraversal(Traversal.INORDER);
System.out.println("\n\nUna vez más INORDER\n");
myTree.forEach( t-> System.out.print(t + " ") );
}
```

Debería obtenerse:

```
Default Traversal
35   20   74   15   22   55   8   27   25
Una vez más INORDER
8   15   20   22   25   27   35   55   74
Una vez más BYLEVELS
35   20   74   15   22   55   8   27   25
Una vez más INORDER
8   15   20   22  25   27   35   55   74
```

Ejercicio 8

8.1) Agregar a la interface e implementar el método

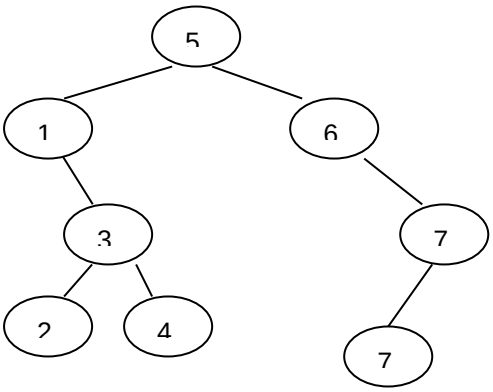
int getOcurrences(T element)

que devuelva la cantidad de ocurrencias del valor solicitado dentro del árbol. **No se debe recorrer innecesariamente** el árbol BSTree que acepta repetidos.

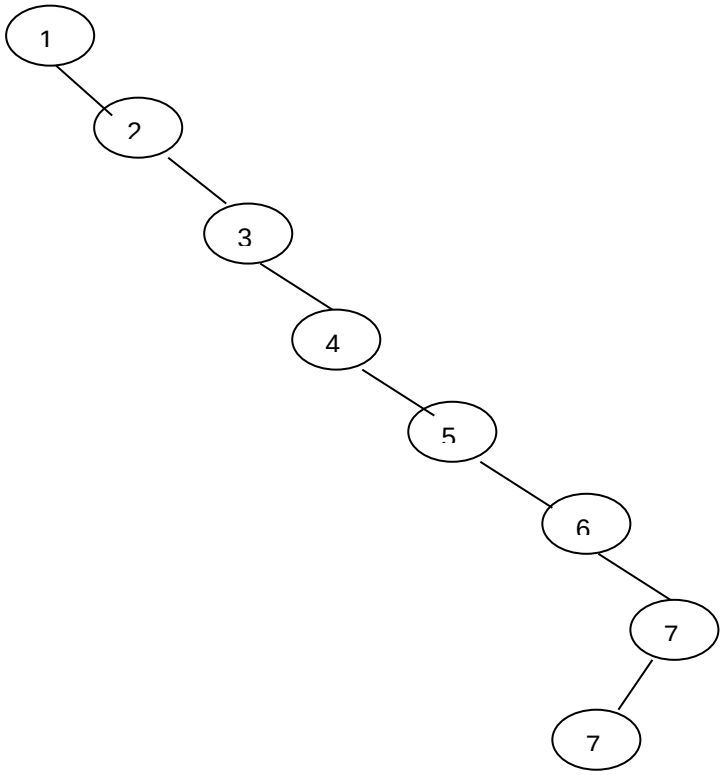
8.2) Calcular la complejidad temporal del método propuesto. Caracterizarlo para cualquier caso (no solo para el peor caso).

Ejercicio 9

En un BST los elementos ocupan un orden respecto de los otros.
Ejemplo 1: si los valores 10, 20, 35, 40, 50, 60, 70, 70 y se hubieran insertado en este orden: 50 60 70 10 70 35 40 20 hubiéramos obtenido:



Ejemplo2: con los mismos valores, si se hubieran insertado en este orden: 10 20 35 40 50 60 70 70 hubiéremos obtenido



En estos ejemplos el mínimo valor presente es 10 y el máximo 70. El mínimo es el valor de 1-esimo, el máximo es el valor N-esimo.

Se quiere obtener para un BST el valor K-esimo. En el ejemplo, el 3-esimo valor es el 35, el 2-esimo es el 20, el 7-esimo el 70, el 8 esimo es el 70, el 10-esimo es null porque ese lugar no está presente, etc.

Agregar a la interface e implementar el método
public T Kesimo(int k)

que devuelva el valor T del nodo que ocupa el lugar indicado por el parámetro o null si el lugar no existiera. Este método debe obtener el lugar por navegación (traversal) y no generando estructuras de datos auxiliares. No recorrer innecesariamente el árbol.

Ejercicio 10

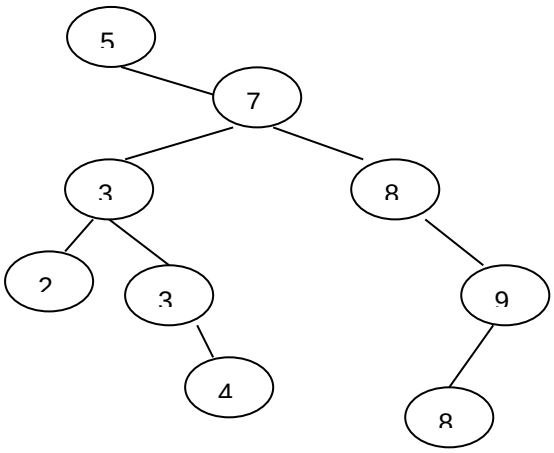
10.1Se tiene un BST que no acepta repetidos (no hace falta cambiar el insert, esta garantizado que eso no ocurre). Agregar a la interface e implementar el método

T getCommonNode(T element1, T element2)

Que devuelve el valor del nodo más cercano que los apunta a ambos, directa o indirectamente. Si no existen las 2 apariciones de element1 y element2 (deben ser diferentes porque no acepta repetidos) devuelve null.

Caso de Uso. Dado este BST sin repetidos

```
myTree = new BST<>();
             myTree.insert(5);
             myTree.insert(70);
             myTree.insert(30);
             myTree.insert(35);
             myTree.insert(20);
             myTree.insert(40);
             myTree.insert(80);
             myTree.insert(90);
             myTree.insert(85);
```




```
n1= 35;
n2= 35;
Integer rta = myTree.getCommonNode (n1, n2);
System.out.println(String.format("common entre %d y %d es %d", n1, n2, rta ) );
```

Se obtendría: `common entre 35 y 35 es null`
porque no hay 2 apariciones de 35

```
n1= 0;
n2= 85;
Integer rta = myTree.getCommonNode (n1, n2);
```

```
System.out.println(String.format("common entre %d y %d es %d", n1, n2, rta ) );
```

Se obtendría: common entre 0 y 85 es null
porque el 0 no está presente

```
n1= 70;
n2= 35;
rta = myTree.getCommonNode(n1, n2);
System.out.println(String.format("common entre %d y %d es %d", n1, n2, rta ) );
```

Se obtendría: `common entre 70 y 35 es 70`

```
n1= 40;
n2= 80;
rta = myTree.getCommonNode(n1, n2);
System.out.println(String.format("common entre %d y %d es %d", n1, n2, rta ) );
```

Se obtendría: `common entre 40 y 80 es 70`

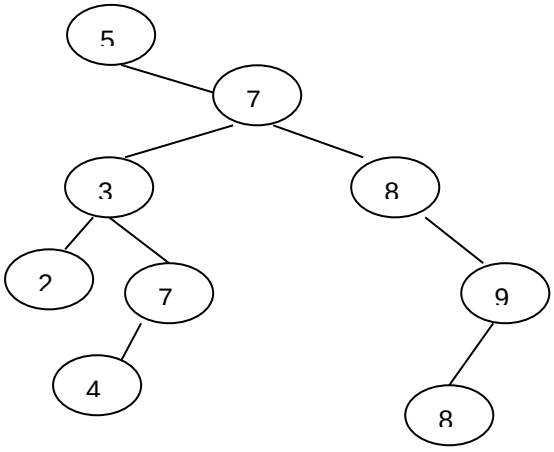
10.2Ahora levantamos la restricción y tiene que funcionar también para BST que acepta repetidos.
Agregar a la interface e implementar el método

T getCommonNodeWithRepeated(T element1, T element2)

Que devuelve el valor del nodo más cercano que los apunta a ambos, directa o indirectamente. Si no existen las 2 apariciones de element1 y element2 devuelve null. Tener en cuenta que si element1 y element2 son iguales, deberá haber por lo menos 2 apariciones de dicho valor para que devuelva diferente de null.

Caso de Uso. Dado este BST sin repetidos

```
myTree = new BST<>();
             myTree.insert(5);
             myTree.insert(70);
             myTree.insert(30);
             myTree.insert(70);
             myTree.insert(20);
             myTree.insert(40);
             myTree.insert(80);
             myTree.insert(90);
             myTree.insert(85);
```




```
n1= 40;
n2= 40;
Integer rta = myTree.getCommonNodeWithRepeated (n1, n2);
System.out.println(String.format("common entre %d y %d es %d", n1, n2, rta ) );
```

Se obtendría: `common entre 40 y 40 es null`
porque no hay 2 apariciones de 40

```
n1= 0;
n2= 85;
Integer rta = myTree.getCommonNodeWithRepeated (n1, n2);
System.out.println(String.format("common entre %d y %d es %d", n1, n2, rta ) );
```

Se obtendría: common entre 0 y 85 es null
porque el 0 no está presente

```
n1= 70;
n2= 70;
rta = myTree.getCommonNodeWithRepeated(n1, n2);
System.out.println(String.format("common entre %d y %d es %d", n1, n2, rta ) );
```

Se obtendría: `common entre 70 y 70 es 70`

```
n1= 40;
n2= 80;
rta = myTree.getCommonNodeWithRepeated(n1, n2);
System.out.println(String.format("common entre %d y %d es %d", n1, n2, rta ) );
```

Se obtendría: `common entre 40 y 80 es 70`

```
n1= 85;
n2= 80;
rta = myTree.getCommonNodeWithRepeated(n1, n2);
System.out.println(String.format("common entre %d y %d es %d", n1, n2, rta ) );
```

Se obtendría: `common entre 85 y 80 es 80`