Segundo Parcial de Estructura de Datos y Algoritmos

Ejer 1	Ejer 2	Ejer 3	Nota
/5	/2.9	/2.1	/10

Duración: 2 horas 10 minutos

Condición Mínima de Aprobación. Deben cumplir estas 2 condiciones:

- Sumar **no menos de cuatro** puntos
- Sumar por lo menos 3 puntos entre el ejercicio 1 y 2.

Muy Importante

Al terminar el examen deberían subir los siguientes 2 archivos, según lo explicado en los ejercicios:

1) Solo las siguientes clases Java: **BST.java** y **SimpleOrDefault.java** según lo pedido. **Cualquier código auxiliar debe estar dentro de esos 2 archivos.**

2) Todos los códigos que les subimos para este examen pertenecen al **package core. Renombrar dicho package con el nombre del usuario de Uds. en campus. Ej: package lgomez**

3) Para todos los ejercicios que no consistan en implementar código Java y pidan calcular complejidades, dibujar matrices, completar cuadros, hacer seguimientos, etc. pueden optar por alguna de estas estrategias:
   a. O completar este documento y subirlo también
   b. O directamente resolverlo en hojas de papel y sacarle fotos (formato jpg, png o pdf) y subir todas las imágenes.

Ejercicio 1

Utilizar el código provisto en este examen, que es un SUBCONJUNTO del **Binary Search Tree paramétrico y acepta repetidos** (similar al implementado en clase).

Se quiere calcular cuáles son los elementos del árbol que están en el intervalo cerrado **[lower, upper] (ambos recibidos como parámetros del método).** Dichos elementos serán devueltos en un HashMap para indicar la cantidad de ocurrencias que han tenido.

Básicamente, el método que deben completar y se encuentra en BST.java es:

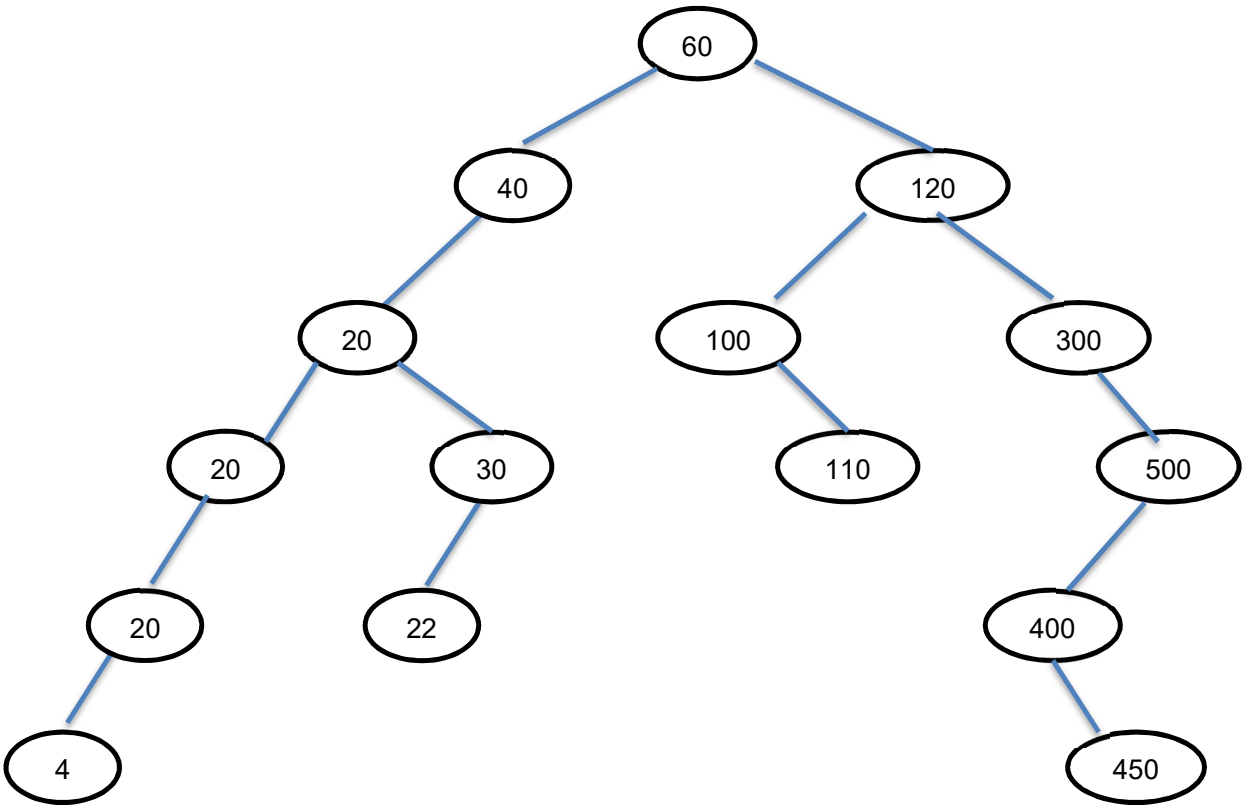
```
@Override
public HashMap<T, Integer> inRange(T lower, T upper) {
   if (lower == null || upper == null)
     throw new RuntimeException("lower and upper cannot be nulls");
  // COMPLETAR
}
```

Sin embargo, se pide implementarlo de una forma especial. **LEER DETENIDAMENTE las indicaciones:**

- Si el intervalo dado por lower y upper no es válido porque lower es más grande que upper, invertirlo y continuar. O sea, no dar error y fixear el problema.
- Si no hay ningún elemento del árbol en el intervalo solicitado, devolver un HashMap creado vacío (no null).
- Los elementos del árbol que no pertenezcan al intervalo NO DEBEN aparecer en el hashmap (ni siquiera con ocurrencias 0).
- **Super importante**: NO recorrer innecesariamente el árbol, es decir NO VISITAR nodos que se saben no conducen a la solución.
- La implementación debe hacerse sin delegar en la clase Node (o sea, directamente en BST. Pueden agregarse métodos private.

Ejemplo

Sea el siguiente BST tree
(pueden crear otros BSTs para verificar correctitud)




```
HashMap<Integer, Integer> rta;
```

```
rta= myTree.inRange(1, 10);
System.out.println(rta); // se obtiene el hashmap con
// key:occurrences {4: 1}

rta = myTree.inRange(23, 91);
System.out.println(rta); // se obtiene el hashmap con
// key:occurrences {40:1, 60:1, 30:1}
```

```
rta = myTree.inRange(23, 400);
System.out.println(rta); // se obtiene el hashmap con
// key:occurrences
// {400:1, 100:1, 40:1, 120:1, 60:1, 300:1, 30:1, 110:1}


rta = myTree.inRange(20, 26);
System.out.println(rta); // se obtiene el hashmap con
// key:occurrences {20:3, 22:1}


rta = myTree.inRange(200, 430);
System.out.println(rta); // se obtiene el hashmap con
// key:occurrences {400:1, 300:1}

rta = myTree.inRange(30, 20);
System.out.println(rta); // se obtiene el hashmap con
// key:occurrences {20:3, 22:1, 30:1}

rta = myTree.inRange(121, 200);
System.out.println(rta); // se obtiene el hashmap  {}


rta = myTree.inRange(900, 800);
System.out.println(rta); // se obtiene el hashmap  {}
```

Ejercicio 2

Utilizar el código provisto en este examen, que es un SUBCONJUNTO del grafo con Lista de Adyacencia implementado en clase.

Definición: **N-regular Simple Graph**

Se dice que **un grafo simple no dirigido** es regular si **TODO vértice** tiene el mismo número de vecinos, o sea, tiene el mismo grado N ≥ 0

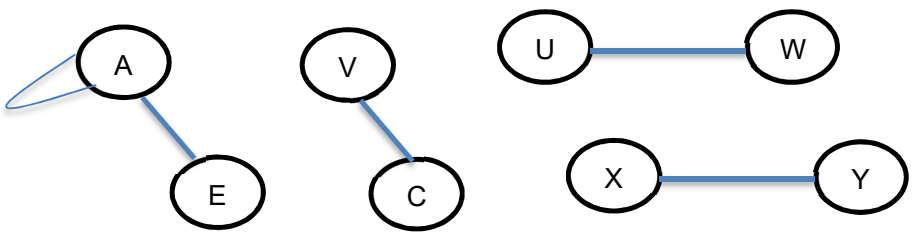
Se pide implementar en la clase **SimpleOrDefault.java** el método **getNRegular()** que **si el grafo es simple y no dirigido, devuelve:**
-1 si no es regular, o bien, devuelve N si es un N-regular graph.

Si el grafo es **Multi o Default (tiene self-loop) lanza RuntimeException**. Nosotros les dimos Multi.java con esa validación, pero el chequeo en SimpleOrDafault lo tienen que hacer Uds.

Caso de Uso A (main1) creado así

```
GraphService<Character, EmptyEdgeProp> g =

new GraphBuilder<Character,
EmptyEdgeProp>().withAcceptSelfLoop(SelfLoop.YES)

.withAcceptWeight(Weight.NO).withDirected(EdgeMode.UNDIRECTED)

.withStorage(Storage.SPARSE).withMultiplicity(Multiplicity.SIMPLE)

.build();
```

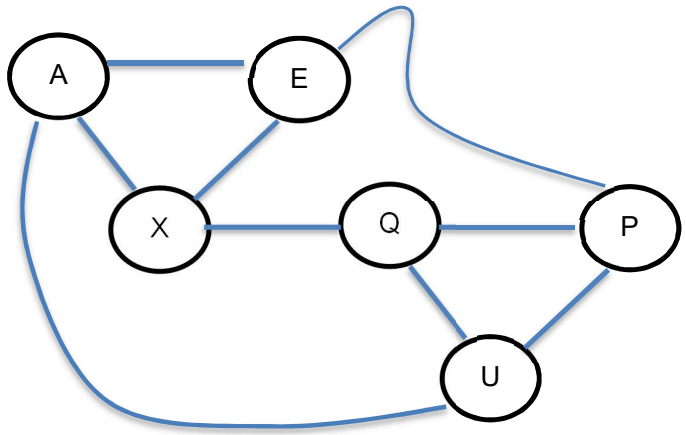
Debe lanzar RuntimeException porque no está definido para grafos default. (aunque no estuviera el lazo en A, si **se lo creó como Self Loop lanza excepción**)



Caso de Uso B (main 2) creado así:

```
GraphService<Character, EmptyEdgeProp> g =

new GraphBuilder<Character,
EmptyEdgeProp>().withAcceptSelfLoop(SelfLoop.NO)

.withAcceptWeight(Weight.NO).withDirected(EdgeMode.UNDIRECTED)

.withStorage(Storage.SPARSE).withMultiplicity(Multiplicity.SIMPLE)

.build();
```

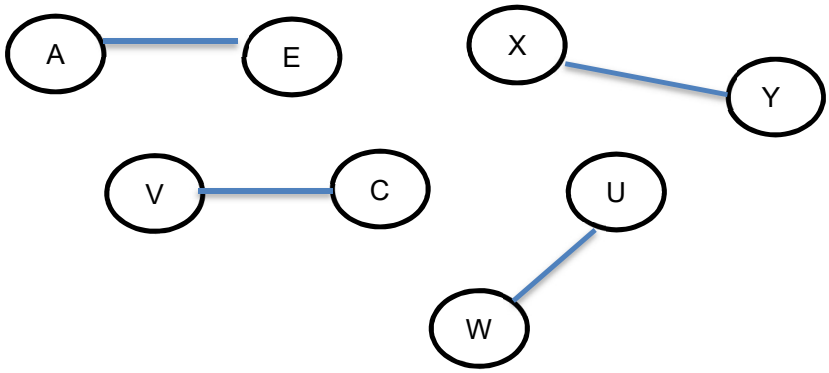
Debe devolver 3



Caso de Uso C (main 3) creado así

```
GraphService<Character, EmptyEdgeProp> g =

new GraphBuilder<Character,
EmptyEdgeProp>().withAcceptSelfLoop(SelfLoop.NO)

.withAcceptWeight(Weight.NO).withDirected(EdgeMode.UNDIRECTED)

.withStorage(Storage.SPARSE).withMultiplicity(Multiplicity.SIMPLE)

.build();
```

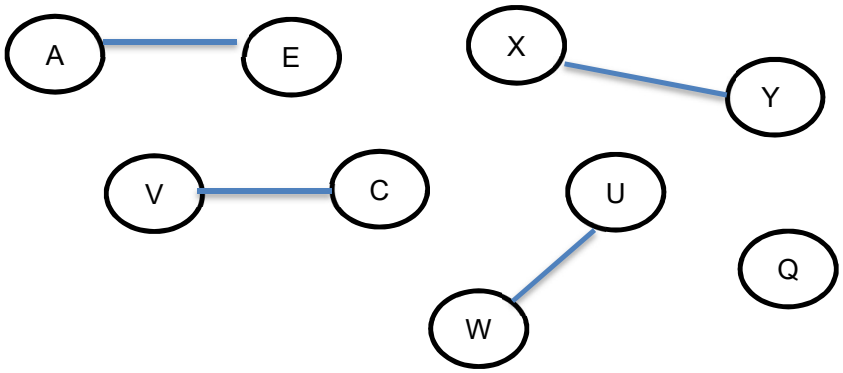
Debe devolver 1



Caso de Uso D (main 4) creado así

```
GraphService<Character, EmptyEdgeProp> g =

new GraphBuilder<Character,
EmptyEdgeProp>().withAcceptSelfLoop(SelfLoop.NO)

.withAcceptWeight(Weight.NO).withDirected(EdgeMode.UNDIRECTED)

.withStorage(Storage.SPARSE).withMultiplicity(Multiplicity.SIMPLE)

.build();
```

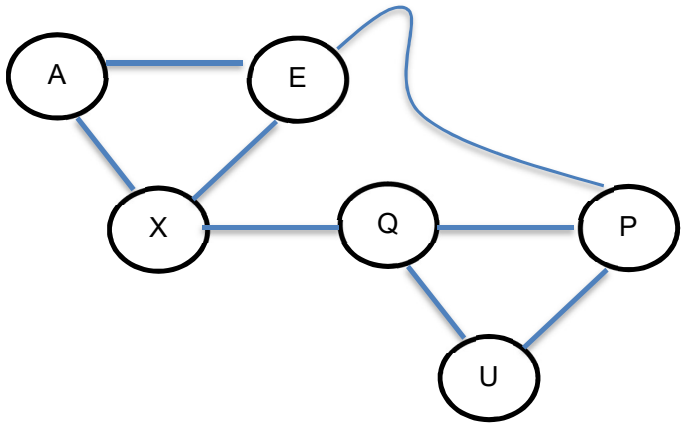
Debe devolver -1 porque aunque es simple y no dirigido, no es regular.



Caso de Uso E (main 5) creado así:

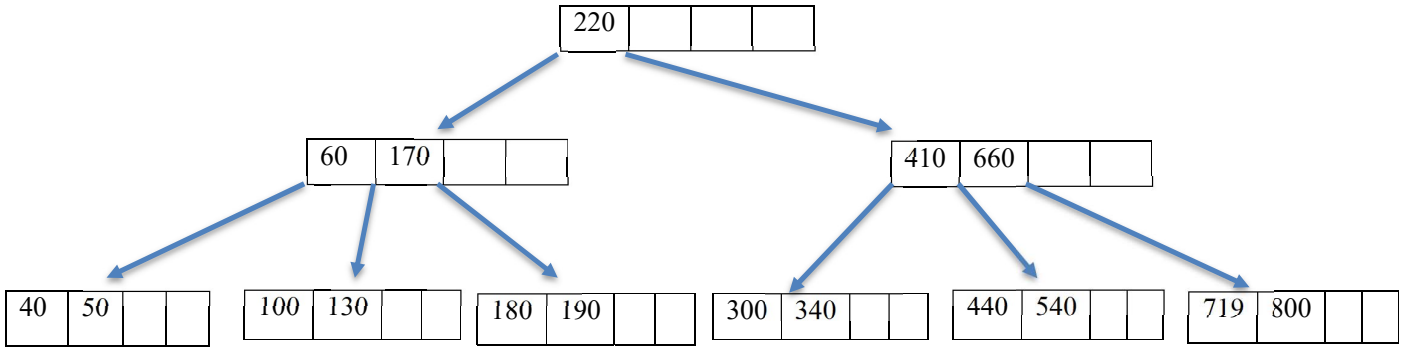
```
GraphService<Character, EmptyEdgeProp> g =

new GraphBuilder<Character,
EmptyEdgeProp>().withAcceptSelfLoop(SelfLoop.NO)

.withAcceptWeight(Weight.NO).withDirected(EdgeMode.UNDIRECTED)

.withStorage(Storage.SPARSE).withMultiplicity(Multiplicity.SIMPLE)

.build();
```

Debe devolver -1 porque aunque es simple y no dirigido, no es regular.



Ejercicio 3

Se tiene el siguiente **árbol B de orden 2**



Mostrar **gráficamente** paso a paso cómo va quedando si se le aplican las **operaciones solicitadas en secuencia**. Mostrar el detalle de lo que ocurre con el árbol al producirse violaciones. Usar siempre **PREDECESOR INORDER. Si no se muestran los pasos intermedios no se considera OK.**

3.1) Borrar el valor 220

3.2) Borrar el valor 719 a partir del árbol obtenido en 3.1

3.3) Borrar el valor 410 a partir del árbol obtenido en 3.2