Segundo Parcial de Estructura de Datos y Algoritmos

Ejer 1	Ejer 2.a	Ejer 2.b	Ejer 3	Nota
/3	/3	/3	/1	/10

Duración: 2 horas 10 minutos

Condición Mínima de Aprobación. Deben cumplir la siguiente condición:

Sumar **no menos de cuatro** puntos entre los ejercicios 1 y 2.

Muy Importante

1. **Al terminar el examen deberían subir los siguientes archivos, según lo explicado en los ejercicios: MultiDirectedGraph.java y BST.java,** según lo pedido. No colocar código auxiliar por fuera de estas clases.

2. Todos los códigos que les subimos para este examen pertenecen al **package default. Colocarlos en un 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

Se provee el código en el archivo **MultiDirectedGraph.java** que implementa la creación de un **grafo multi dirigido sin self-loops** (Bajarlo de Campus).

Se pide agregar el método

MultiDirectedGraph<V,E> transposeSummarize(BiFunction<E, E, E> summarizer)

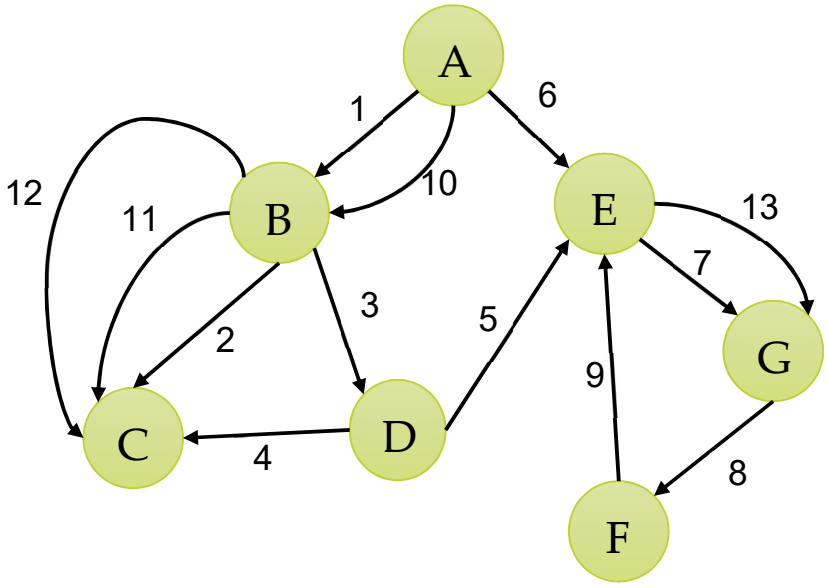
Que devuelve el grafo transpuesto donde todos los ejes se encuentran invertidos y si hubiera más de un eje entre 2 nodos $X \rightarrow Y$, se deberán sumarizar usando la función summarizer provista. De esta manera el grafo original que podría ser un **MultiGrafo pasará a estar transpuesto y simplificado dado que no habrá más de un eje dirigido entre 2 nodos en el grafo resultante.**

Muy Importante:

- **No se pueden agregar miembros de datos a las clases provistas**.
- **No se puede modificar el código entregado.**

Ejemplo 1

Dado el grafo y con la BiFunction que corresponde a la suma de los valores



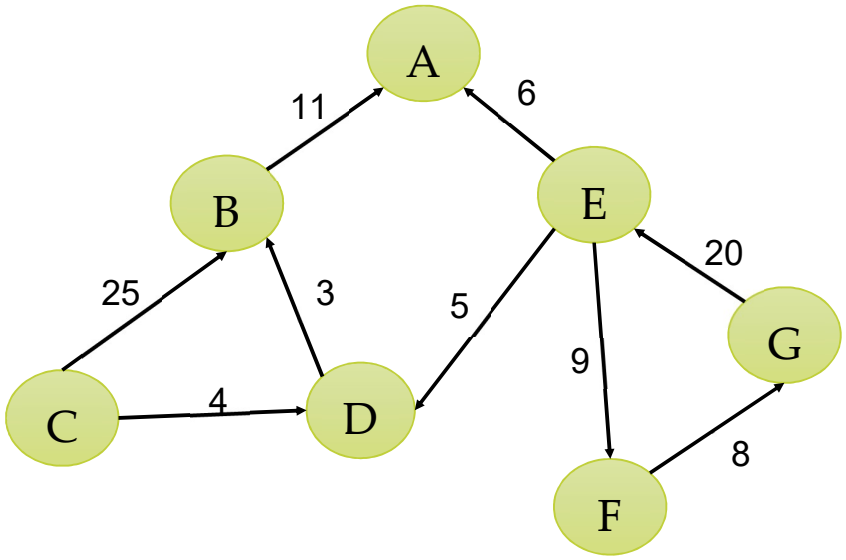
Al imprimir los vértices del grafo original el resultado es:

```
A
B
C
D
E
F
G
```

Y los ejes:

```
A -> B[1]
A -> E[6]
A -> B[10]
B -> C[2]
B -> D[3]
B -> C[11]
B -> C[12]
D -> C[4]
D -> E[5]
E -> G[7]
E -> G[13]
F -> E[9]
G -> F[8]
```

Después de llamar a **transposeSummarize** se debe devolver el siguiente grafo:



Donde al imprimir los vértices se devuelve:

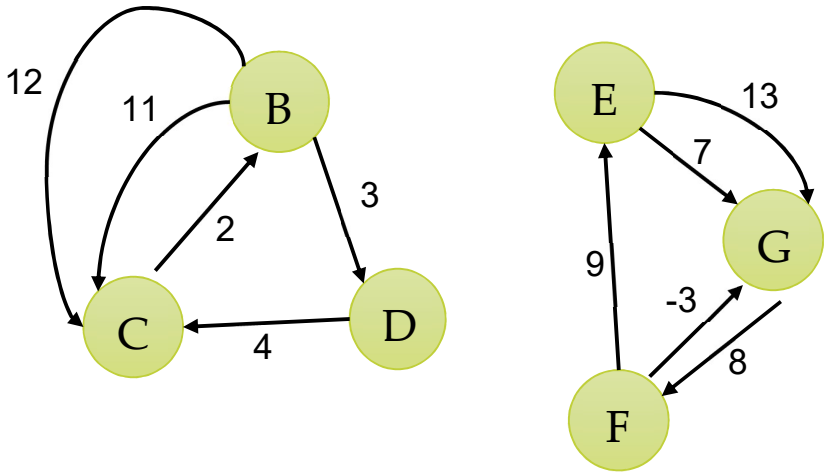
```
A
B
C
D
E
F
G
```

y los ejes:

```
B -> A[11]
C -> B[25]
C -> D[4]
D -> B[3]
E -> A[6]
E -> D[5]
E -> F[9]
F -> G[8]
G -> E[20]
```

Ejemplo 2

Dado el grafo y con la BiFunction que corresponde a la suma de los valores



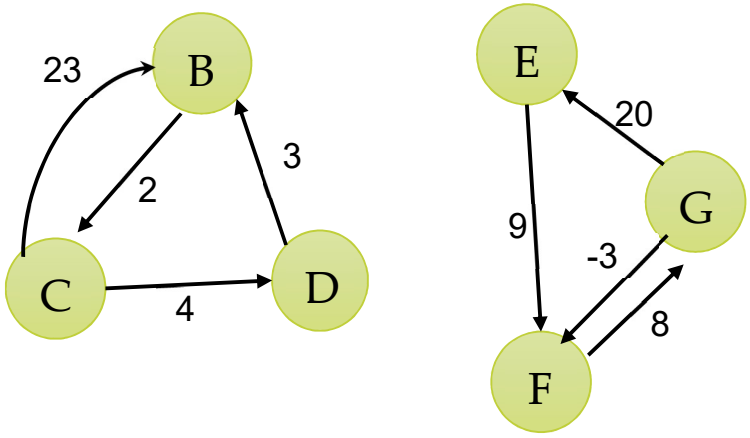
Al imprimir los vértices del grafo original el resultado es:

```
B
C
D
E
F
G
```

y los ejes:

```
B -> C[11]
B -> C[12]
B -> D[3]
C -> B[2]
D -> C[4]
E -> G[7]
E -> G[13]
F -> G[-3]
F -> E[9]
G -> F[8]
```

Después de llamar a **transposeSummarize** se debe devolver el siguiente grafo:



Donde al imprimir los vértices se devuelve:

```
B
C
D
E
F
G
```

y los ejes:

```
B -> C[2]
C -> B[23]
C -> D[4]
D -> B[3]
E -> F[9]
F -> G[8]
G -> E[20]
G -> F[-3]
```

Ejercicio 2

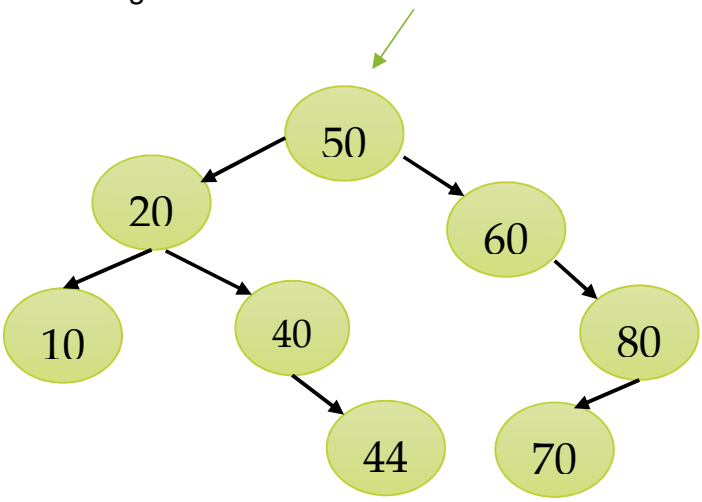
Se provee el código en el archivo ***BST.java*** (bajarlo de Campus) que implementa un BinarySearchTree simple no balanceado**.** Se provee el método insert.

Muy Importante para ambos puntos:

- **No se pueden agregar miembros de datos a las clases provistas**.
- **No se puede modificar el código entregado.**

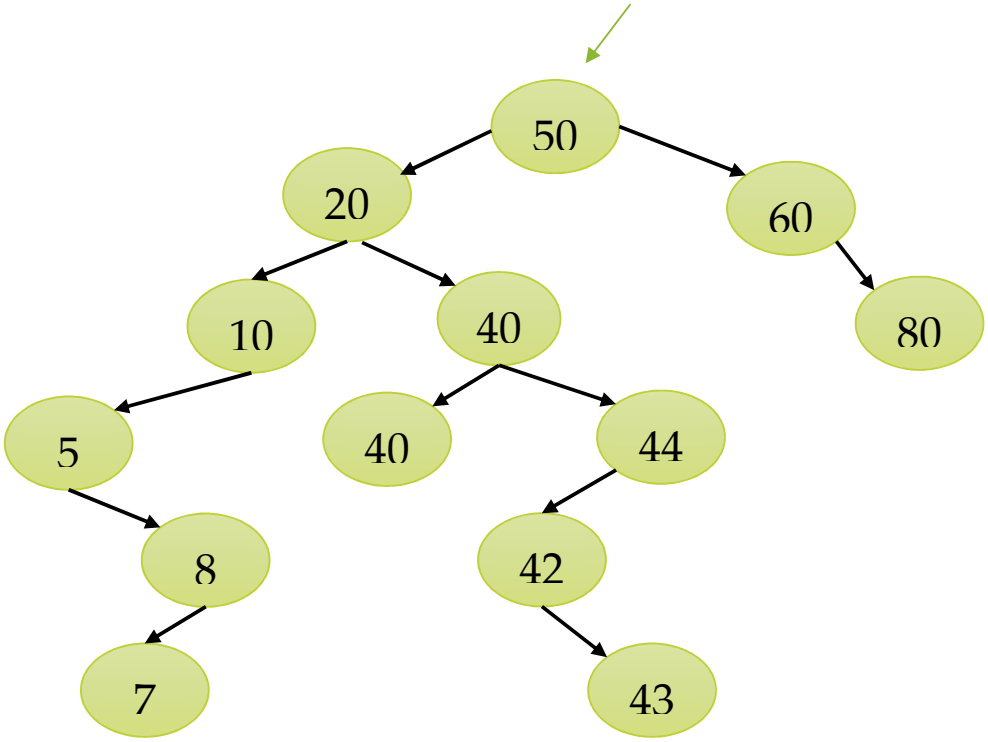
2.a) se define el diámetro de un árbol como la longitud del camino simple más largo (en cantidad de ejes) entre cualquier par de nodos. El camino más largo no necesariamente pasa por la raíz.

Ejemplo 1:
Para el siguiente árbol:



El **diámetro es de 6** que es el camino entre los nodos 44 y 70. En este caso el diámetro pasa por el root.

Ejemplo 2:



El **diámetro es de 8** que es el camino entre los nodos 7 y 43. En este caso el nodo común a ambos es el 20 y no pasa por la raíz.

Se pide: implementar el método int getDiamater() que devuelva el valor del diámetro del árbol.

2.b) Implementar el siguiente método: **ArrayList<T> getDiameterPath()** donde se devuelve los valores de los nodos que corresponden al camino simple que corresponden al diámetro: desde un extremo hacia el otro y comenzando por el nodo de menor valor.

Es decir, parado en el nodo en común entre los nodos más alejados primero se deben retornar los nodos del subárbol izquierdo empezando por el nodo hoja, luego el nodo común y finalmente los nodos del subárbol derecho siguiendo por el primero adyacente al nodo común y así hasta el nodo hoja más alejado del primero.
En caso de haber más de un camino simple correspondiente al diámetro, devolver cualquiera de ellos.

En los ejemplos anteriores se debe devolver el ArrayList con el siguiente contenido:

para el árbol del ejemplo 1:
44, 40, 20, 50, 60, 80, 70

para el árbol del ejemplo 2:
7, 8, 5, 10, 20, 40, 44, 42, 43
Se marca el nodo común como referencia.

Ejercicio 3

Se tiene un hashing implementado con Open Addressing (también llamado Closed Hashing o **sin chaining**) y teniendo en cuenta la siguiente configuración:

- initialLookUpSize= 10;
- Threshold= 0.75
- Resize +0.5 (o sea se aumenta en un 50 % de su tamaño)
- La función de pre hash es la identidad. Las claves son enteras.
- Para la resolución de colisiones se utiliza **Resaheo Cuadrático (quadratic probing)**. Manejo circular de la estructura subyacente
- Como siempre: **primero se coloca el elemento** y, si hace falta, **luego se chequea si se superó el Threshold**.

Hasta ahora se tiene

pos	(key, value)	Estado (ocupado, baja lógica, baja física)
0	<30, “P”>	Ocupado
1		Baja Fisica
2	<22, “E”>	Ocupado
3		Baja física
4	<14, “A”>	Ocupado
5	<25, “D”>	Ocupado
6	<36, “B”>	Ocupado
7		Baja física
8	<18, “C”>	Ocupado
9	<39, “Z”>	Ocupado

La próxima operación es:

myLookUp.insert(**8,"T"**);

Se pide:

- Señalar el proceso para obtener la ranura correspondiente, es decir, **indicar las ranuras que fueron chequeadas hasta que se colocó el elemento pedido.**

- **Mostrar la tabla de hashing final.**