Notas:

- Tu solución debe compilar sin errores para que el examen sea corregido. Asegúrate de completar o comentar el código que consideres necesario antes de entregarlo.
- A la hora de evaluar tu solución, se tendrá en cuenta la corrección, la eficiencia y la claridad.
- Puedes utilizar el material en la página de la asignatura del campus virtual de la UMA. El uso de cualquier otro material será motivo de suspenso.
- El enunciado del examen consta de varios ficheros disponibles a través del CV:
    - Un ejemplo de ejecución del algoritmo (kruskal.pdf).
    - Biblioteca de estructuras de datos y programas de prueba básica.
    - Plantilla de los programas Java y Haskell a completar.

Un árbol de recubrimiento o de expansión de un grafo conexo es un subconjunto de aristas del mismo que conforman un árbol (un subgrafo conexo y sin ciclos) que incluye todos los vértices del grafo original. El algoritmo de Kruskal (Joseph Kruskal, publicado en 1956) encuentra de forma eficiente un árbol de recubrimiento de peso total mínimo para un grafo con pesos. En caso de que el grafo conste de varias componentes conexas, este mismo algoritmo encontrará un bosque de recubrimiento mínimo, es decir, un árbol de recubrimiento mínimo para cada componente conexa.

La idea intuitiva del algoritmo consiste en partir del grafo sin aristas e ir añadiendo incrementalmente aquellas que conforman el bosque de recubrimiento mínimo. Al ir añadiendo estas aristas, se irán conectando vértices que previamente estaban desconectados, formándose progresivamente los distintos árboles de recubrimiento mínimos. Para ello, el algoritmo va considerando las aristas del grafo ordenadas ascendentemente por sus pesos. Para esto, utilizaremos una cola de prioridad PQ, que inicialmente contendrá todas las aristas del grafo. Representaremos el bosque de recubrimiento mínimo que queremos calcular como un conjunto T (inicialmente vacío) de aristas con peso. Para saber si una nueva arista conectaría vértices previamente desconectados, mantenemos un diccionario DICT, que representa la conectividad entre vértices que resulta al ir añadiendo aristas al bosque de recubrimiento T. Este diccionario se inicializa de forma que cada vértice v del grafo esté asociado consigo mismo (v->v). A continuación, iremos procesando cada una de las aristas de PQ, según el orden ascendente de sus pesos, de forma que:

- Si la arista une dos vértices que no están conectados aún según DICT, se actualiza el diccionario DICT para representar que éstos pasan a estar conectados y la arista se añade al bosque de recubrimiento mínimo T.
- Si la arista une dos vértices que ya están conectados según DICT, no se hace nada con ésta.

En el momento en que no quedan más aristas por procesar en PQ, tendremos en T un árbol de recubrimiento de peso mínimo por cada componente conexa del grafo de partida.

Para saber si dos vértices han sido ya conectados, comprobamos si ambos tienen el mismo **representante** en el diccionario DICT. Para obtener el representante de un vértice basta con partir de él y seguir repetitivamente la secuencia de asociaciones (clave -> valor) en el diccionario hasta encontrar un vértice asociado con sigo mismo; éste último será el representante del vértice original. Para representar la nueva conexión que resulta como resultado de añadir una nueva arista a T, basta con añadir al diccionario DICT una asociación entre el representante de uno de los vértices de la arista y el otro vértice, de forma que los vértices en los árboles fusionados pasan a tener el mismo representante.

Además, necesitaremos una implementación de grafos no dirigidos con pesos, implementados mediante un diccionario en el que, a cada vértice v del grafo, se le asocia otro diccionario, en el que a cada uno de los sucesores de v se le asocia el peso de la arista entre ambos. Para que esta implementación del grafo funcione de forma adecuada, es necesario que todos los vértices del grafo sean claves del diccionario "principal". Es decir, cuando se añade un vértice v al grafo, se añade una asociación (v, diccionario vacío) en el diccionario "principal". Cuando se inserte una arista v1-v2 con peso w en el grafo, en el diccionario asociado al vértice v1 se añade una asociación v2->w, donde v2 sería la clave y w el valor.

Java

Descarga del campus virtual el archivo comprimido que contiene el proyecto Eclipse para resolver el problema y comprobar tu solución. Completa las definiciones de métodos de los ficheros

- dataStructures.graph.DictionaryWeightedGraph.java
- Kruskal.java

Estos dos ficheros son los que debes subir a la tarea del campus virtual.

La clase DictionaryWeightedGraph<V,W> implementa la interfaz WeightedGraph<V,W> utilizando una estructura Dictionary<V,Dictionary<V,W>>, tal como se ha explicado previamente. Implementa los siguientes métodos:

J.1) (0.25 puntos) El método void addVertex(V v) que inserta un vértice en el grafo.

J.2) (0.25 puntos) El método void addEdge(V u, V v, W w) que inserta una arista en el grafo. Los vértices deben estar en el grafo; en otro caso debe lanzar una excepción de tipo GraphException.

J.3) (0.5 puntos) El método Set<Tuple2<V,W>> successors(V v) que devuelve un conjunto con los sucesores del vértice v en el grafo junto con los pesos correspondientes. Si v no está en el grafo, se lanzará una excepción de tipo GraphException.

J.4) (0.5 puntos) El método Set<WeightedEdge<V,W>> edges() que devuelve un conjunto con todas las aristas del grafo.

J.5) (0.75 puntos) Completa la clase WE<V,W> que implementa la interfaz WeightedEdge<V,W> y representa una arista con peso. Ten en cuenta que dos aristas son iguales si conectan el mismo par de vértices con el mismo peso. Para definir compareTo, ten en cuenta que el orden entre aristas viene dado únicamente por el orden de sus pesos.

J.6) (1.5 puntos) El método Set<WeightedEdge<V,W>> kruskal(WeightedGraph<V,W> g) que devuelve un bosque de recubrimiento mínimo, con un árbol por cada una de las componentes conexas del grafo g, utilizando el algoritmo de Kruskal que hemos descrito.

La clase KruskalTest contiene ejemplos de grafos de prueba junto con la salida esperada para comprobar el funcionamiento de los métodos implementados. Aunque obtener la misma salida no garantiza la corrección total de tu código, puede ayudar a corregir su funcionamiento y entender el uso de las distintas funciones.

Sólo para alumnos a tiempo parcial

J.7) (1.25 puntos) Define un método

Set<Set<WeightedEdge<V,W>>> kruskals(WeightedGraph<V,W> g)

que devuelva un conjunto de árboles de recubrimiento mínimos (un subconjunto para cada una de las componentes conexas del grafo argumento), de forma que las aristas que pertenecen a componentes conexas distintas aparezcan en subconjuntos distintos en el resultado.

Haskell

Descarga el archivo comprimido que contiene los ficheros necesarios para resolver el problema y comprobar tu solución. Completa las definiciones de métodos de los ficheros:

- DataStructures.Graph.DictionaryWeightedGraph.hs
- Kruskal.hs

Estos son los dos únicos ficheros que debes subir a la tarea del campus virtual.

El módulo DataStructures.Graph.DictionaryWeightedGraph implementa un grafo no dirigido con pesos utilizando una estructura

```
data WeightedGraph a w  = WG (D.Dictionary a (D.Dictionary a w))
```

tal como se ha explicado previamente. Implementa las siguientes funciones:

H.1) (0.5 puntos) La función

```
addVertex :: (Ord a) => WeightedGraph a w -> a -> WeightedGraph a w
```

que inserta un vértice en el grafo.

H.2) (0.5 puntos) La función

```
addEdge :: (Ord a, Show a) => WeightedGraph a w -> a -> a -> w -> WeightedGraph a w
```

que inserta una arista con peso en el grafo. Los vértices deben estar en el grafo; en otro caso debe producirse un error.

H.3) (0.5 puntos) La función

```
successors :: (Ord a, Show a) => WeightedGraph a w -> a -> [(a,w)]
```

que devuelve una lista con los sucesores de un vértice en el grafo junto con el peso correspondiente. Si el vértice no está en el grafo, se debe producir un error.

H.4) (0.5 puntos) La función

```
edges :: (Eq a, Eq w) => WeightedGraph a w -> [WeightedEdge a w]
```

que devuelve una lista con todas las aristas con peso del grafo.

H.5) (1.75 punto) La función

```
kruskal :: (Ord a, Ord w) => WeightedGraph a w -> [WeightedEdge a w]
```

que devuelve un bosque de recubrimiento mínimo, con un árbol para cada una de las componentes conexas del grafo dado como argumento, utilizando el algoritmo de Kruskal descrito previamente.

El fichero KruskalTest.hs contiene ejemplos de grafos de prueba junto con la salida esperada para comprobar el funcionamiento de los métodos implementados. Aunque obtener la misma salida no garantiza la corrección total de tu código, puede ayudar a corregir su funcionamiento y entender el uso de las distintas funciones.

Sólo para alumnos a tiempo parcial

H.6) (1.25 puntos) Define una función

```
kruskals :: (Ord a, Ord w) => WeightedGraph a w -> [[WeightedEdge a w]]
```

que devuelva una lista de árboles de recubrimiento mínimos (una sublista para cada una de las componentes conexas del grafo argumento), de forma que las aristas que pertenecen a componentes conexas distintas aparezcan en sublistas distintas en el resultado.