# Práctica 1: Árboles generales

## Normas

- Este enunciado va acompañado de un fichero ZIP que contiene el código necesario para hacer la práctica. Los *tests* que acompañan a las prácticas son orientativos y pueden ser ampliados por los estudiantes si lo consideran necesario.
- El beneficio que obtiene el estudiante al realizar la práctica es mejorar sus habilidades como programador. Estas habilidades serán evaluadas en el examen final de la asignatura.
- El código proporcionado está preparado para ejecutarse en Java 17.0.1 con la versión de JUnit 5.8.2. No obstante, el estudiante es libre de utilizar las versiones de Java y JUnit que considere oportuno, siempre y cuando el código de la práctica se pueda ejecutar correctamente con las versiones anteriormente citadas.

## Ejercicio 1: Operaciones sobre árboles n-arios

Para aumentar la funcionalidad de los árboles n-arios se desea disponer de mecanismos para obtener su profundidad, su grado y un iterador a la frontera compuesta por todas sus hojas. Para ello se pide implementar las siguientes clases cuya interfaz ya está en el paquete `Tree`:

- `DepthCalculator`
- `DegreeCalculator`
- `FrontierIterator`

Para hacer este ejercicio solo deben tocarse los ficheros: `DepthCalculator.java`, `DegreeCalculator.java` y `FrontierIterator.java`.
Los *tests* incluidos en el material suministrado ayudan a entender cómo deben funcionar estas clases.

## Ejercicio 2: Implementación de un montículo

En la clase `HeapPriorityQueue` se implementa parcialmente la funcionalidad de un montículo (*heap*) utilizando un Array para representar el árbol internamente. Se pide implementar los siguientes métodos:

- `void upHeap(int j)`: este método recibe como parámetro el índice *j* de un nodo *p* del montículo. El objetivo del método es reordenar el nodo *p* para que el montón cumpla la propiedad de *heap-order*. En particular, este método debe recolocar el nodo *p* realizando intercambios con su nodo padre hasta que el padre tenga una clave menor o igual que el nodo *p*. Este proceso es conocido como *up-heap bubbling*.
- `void downHeap(int j)`: este método recibe como parámetro el índice j de un nodo p del montículo. El objetivo del método es reordenar el nodo p para que el montón cumpla la propiedad de *heap-order*. En particular, este método debe recolocar el nodo *p* realizando intercambios con uno de sus dos hijos (específicamente, con aquel cuya clave es menor) hasta que ninguno de los dos hijos tenga una clave menor que el nodo *p*. Este proceso es conocido como *down-heap bubbling*.

Para visualizar estos procesos, se recomienda utilizar la herramienta visual que se pueda encontrar en este enlace: https://www.cs.usfca.edu/~galles/visualization/Heap.html

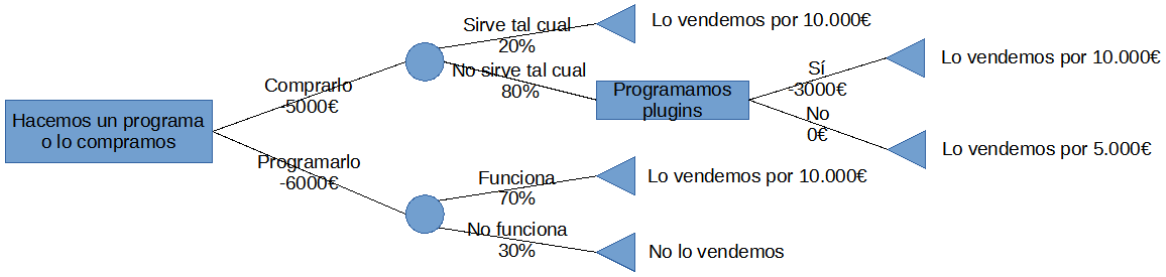
En el fichero `HeapTests` se proporcionan algunos *tests* para comprobar la funcionalidad del montículo.

## Ejercicio 3: Diagrama de árbol de decisión

Un árbol de decisión es un árbol n-ario que representa los posibles resultados de una serie de decisiones relacionadas. Un árbol permite comparar las posibles acciones según sus costos, probabilidades y beneficios. Los árboles de decisión se usan para analizar ideas o para crear algoritmos que escojan la mejor opción.

Básicamente, el comportamiento de un árbol de decisión viene determinado por las siguientes reglas:

- La raíz de un árbol de decisión siempre es un nodo de decisión. Luego, a cualquier nodo de un árbol de decisión se le puede añadir un nodo de tipo decisión, uno de tipo probabilidad o uno de tipo terminal.
- De un nodo tipo decisión salen aristas con una cadena y un coste/beneficio asociado hacia sus descendientes.
- De un nodo de tipo probabilidad salen aristas con una cadena y una probabilidad asociada hacia sus descendientes.
- Los nodos de tipo terminal tienen un coste/beneficio asociado.
- A cada hijo de un nodo decisión se le puede pedir su coste/beneficio. Para calcular el resultado debe realizarse la suma de las cantidades desde los nodos terminales y el producto por las probabilidades hasta llegar a la decisión en cuestión. Un nodo de probabilidad propaga hacia su padre la suma de las cantidades que le llegan, mientras que un nodo de decisión propaga hacia su padre el máximo de los valores que le llegan.




Por ejemplo, sobre el diagrama adjunto las siguientes preguntas arrojan el siguiente resultado:

- Qué esperanza de beneficio tiene programar los plugins: 7.000 € (10.000 - 3000)
- Qué esperanza de beneficio tiene no programar los plugins: 5.000 € (5.000 - 0)
- Qué esperanza de beneficio tiene hacer el programa: 1.000 € (10.000 * 0.7 + 0 * 0.3 - 6000)
- Qué esperanza de beneficio tiene comprar el programa: 2.600 € (max(5.000 - 0, 10.000 - 3000) * 0.8 + 10.000 * 0.2 – 5000)

Para aprender más sobre los árboles de decisión se recomienda la lectura de la siguiente página.

Desarrollar una clase que permita construir árboles de decisión. La clase desarrollada deberá cumplir la interfaz `DecisionTree` suministrada en el fichero ZIP y deberá pasar los *tests* adjuntos.

## Observaciones generales

Está prohibido modificar cualquier parte de la interfaz pública de cualquiera de las clases proporcionadas.

Los *tests* se proporcionan a modo de ejemplo. Que la práctica pase los *tests* proporcionados no quiere decir que esté correctamente resuelta. En general, se recomienda ampliar los *tests* para verificar el correcto funcionamiento de todo lo que se implemente.