Normas del examen

- En todos los ejercicios se valorará la eficiencia de las estructuras de datos utilizadas y del código implementado.
- Si un código resuelve el problema de forma no eficiente se podrá considerar como no válido, aunque la salida sea correcta.
- Para comenzar a resolver el examen, descarga el fichero con el código de apoyo del Aula Virtual, descomprime ese fichero y copia el contenido de la carpeta src dentro de la carpeta src de tu proyecto.
- La entrega se basará en un único fichero ZIP que contendrá la carpeta src con todo el código implementado.
- Puedes implementar tantas clases y métodos como necesites, pero sin modificar las cabeceras de los métodos propuestos.
- No se permite la utilización de ningún tipo de apoyo, ni prácticas realizadas previamente. Cualquier intento de utilización de este tipo de herramientas implicará el suspenso automático de la convocatoria completa y las medidas disciplinarias oportunas.

Ejercicio 1 [3 puntos]

Un montón o *heap* es una estructura de datos basada en un árbol que satisface la siguiente propiedad: Si A es un nodo padre de B, entonces *key*(A) se ordena con respecto a *key*(B), aplicándose el mismo orden en todo el montón. Esto es, o bien las claves de los nodos padre son siempre mayores o iguales que las de los hijos y la clave más alta está en el nodo raíz (montón máximo o *maximum heap*) o las claves de los nodos padre son menores o iguales que las de los hijos y la clave más baja está en el nodo raíz (montón mínimo o *minimum heap*).

Se pide implementar un *minimum heap* basado en un árbol en forma de array. La raíz del árbol se encuentra en la posición 0 del array, y los hijos de cada nodo se encuentran en la posición n * i + k, siendo n el tamaño máximo para el array, i el nodo del que buscamos el hijo y k el hijo que buscamos. Se desea implementar la funcionalidad en la clase `HeapTree` que se encuentra en Aula Virtual junto al material del examen. En concreto, se debe implementar la funcionalidad de los métodos:

a) Implementar el método `insert`, que insertará un elemento E en el heap, doblando el tamaño del array si este está lleno. El método debe dejar el heap en un estado de forma que, al extraer el siguiente elemento, sea el que corresponda.
b) Implementar el método `subirEnHeap`, que será el encargado de mantener el heap ordenado, colocando en la raíz del árbol el nodo mínimo cuando un nodo se inserte.
c) Implementar el método `remove`, que eliminará un elemento E en el heap. El método debe dejar el heap en un estado de forma que, al extraer el

siguiente elemento, sea el que corresponda. Este método devolverá el elemento que se ha eliminado y en su posición colocará el último elemento en el heap en ese momento (esta circunstancia se reparará con el método `bajarEnHeap`).

d) Implementar el método `bajarEnHeap`, que será el encargado de mantener el heap ordenado cuando un nodo se elimine. Debe encargarse de mover el elemento que se pasa como argumento a la posición que le corresponda, subiendo en el *heap* el menor de sus hijos hasta que no tenga ningún elemento menor entre sus hijos.

Ejercicio 2 [5 puntos]

La dificultad del juego Hollow Knight está haciendo que los desarrolladores implementen *mods* que permitan personalizar el juego para disponer de la información más importante de manera inmediata. Nuestro objetivo es implementar un *mod* que nos permita consultar nuestro inventario de manera rápida y, además, almacenar las zonas del mapa junto con sus conexiones. De cada objeto se almacena información sobre su nombre (que lo identifica), su descripción y el número de unidades de las que disponemos, que se modificará constantemente al utilizarlo o adquirir más. De cada zona se almacena su nombre, su nivel de dificultad que estará entre 0 y 100 y el número de enemigos que hay en esa zona. Las zonas están conectadas formando una jerarquía de manera que siempre se comienza desde la primera zona, y esta te permite ir a una serie de zonas determinadas. Al superar una de estas zonas, se abre la posibilidad de ir a nuevas zonas, y así sucesivamente. Se pide:

a) Definir las estructuras de datos adecuadas para almacenar la información necesaria y minimizar la complejidad de las operaciones indicadas.

b) Implementar los siguientes métodos, cuya cabecera está definida en la clase `HollowKnight`:

- `newItem`: Crea un nuevo elemento que se puede almacenar, pero sin existencias en el inventario del jugador. Lanza una excepción con el mensaje "`Ya existe un elemento con ese nombre`" en caso de que ya exista un elemento con el mismo nombre.
- `addItem`: Añade una unidad al elemento indicado en el parámetro. En caso de no existir ningún elemento con ese nombre, se lanza una excepción con el mensaje "`No puedes anadir nuevos elementos si no han sido previamente creados`"
- `useItem`: Elimina una unidad del elemento indicado en el parámetro. Si el elemento no existe, lanza una excepción con el mensaje "`El elemento que quieres utilizar no existe`". En caso de no disponer de unidades para gastar, se lanza una excepción con el mensaje "`No te quedan NAME,`

`no puedes utilizarlo`", donde `NAME` es el nombre del elemento.

- `addArea`: Añade una nueva zona de juego con el nombre, número de enemigos y dificultad indicada en los parámetros.

c) Implementar el método `maxEnemies`, cuya cabecera está definida en la clase `HollowKnight`. Suponiendo que el jugador comienza en la primera zona, este método calcula el número máximo de enemigos que derrotaría el jugador siguiendo cualquiera de los caminos de la jerarquía de zonas.

d) Implementar el método `countSteps`, cuya cabecera está definida en la clase `HollowKnight`. En este método, dada una de las zonas almacenadas, se pide indicar cuántas zonas tiene que recorrer el jugador desde la zona inicial para llegar a ella.