Estructuras de Datos

Grupos B y D

Convocatoria ordinaria 2021-2022

Recuerda poner tu nombre y tu usuario del juez en un comentario al principio de todos los archivos que entregues

Ejercicio 1 [2,5 puntos]

En ocasiones se desea penalizar una secuencia de elementos de una cola enviándolos detrás del último elemento de la misma. Para ello vamos a añadir la operación

```
void penalizaSecuencia(int inicio, int fin);
```

a la implementación del TAD `Cola` basada en nodos enlazados vista en clase, que traslada la secuencia compuesta por los elementos de la cola que se encuentran entre la posición `inicio` y la posición `fin`, ambas incluidas, detrás del último elemento de la cola, respetando la disposición relativa de los elementos trasladados. La primera posición de la cola es la 0, la segunda la 1 y así sucesivamente.

Por ejemplo, dada la siguiente cola de números enteros, donde el 15 es el primer elemento y el 21 es el último elemento

15	3	5	32	21

el resultado de penalizar la secuencia entre las posiciones 2 y 3 es el siguiente:

15	3	21	5	32

Si las posiciones están fuera del rango posible, la secuencia involucra al último elemento de la cola, o la posición `fin` es menor que la posición `inicio`, la operación no surtirá ningún efecto.

En la implementación de esta nueva operación no se permiten utilizar, ni directa, ni indirectamente, operaciones de manejo de memoria dinámica (`new`, `delete`), ni tampoco realizar asignación alguna entre contenidos de los nodos. Aparte de implementar la operación, debes determinar justificadamente su complejidad.

Ejercicio 2 [2,5 puntos]

La empresa `FunnyGames` nos ha encargado desarrollar un programa que evalúe la *jugabilidad* de sus juegos de supervivencia. Todos son juegos en primera persona, en los que el jugador comienza el juego con una cantidad inicial de energía. Así mismo, involucran tres tipos de acciones diferentes: luchas, superación de obstáculos, y consumo de alimentos. Las luchas pueden ganarse (en este caso no se consume energía), o perderse (este caso sí supone una pérdida de energía). La superación de obstáculos siempre supone una pérdida de energía. Por último, el consumo de alimentos permite recuperar energía. Si tras llevar a cabo una cierta acción el jugador se queda sin energía, pierde la partida. Si llega al final del juego, gana la partida.

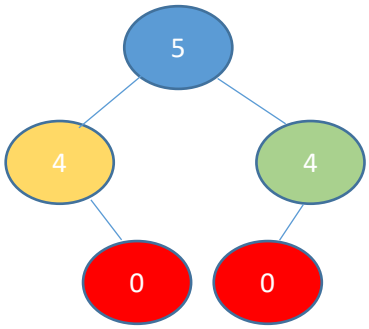
Un juego es *jugable* para una cierta energía inicial si puede haber partidas que se ganan y partidas que se pierden.

Para evaluar la jugabilidad para una cierta energía inicial, representamos los juegos mediante árboles binarios de enteros no negativos, formados por los siguientes tipos de nodos:

- Nodos *lucha*. Nodos que representan luchas. Son aquellos que tienen tanto un hijo izquierdo como un hijo derecho. El hijo izquierdo representa la continuación del juego si la lucha se gana, y el hijo derecho la continuación del juego si la lucha se pierde. El valor del nodo debe ser un número positivo, y es la cantidad de energía que el jugador pierde en caso de perder la lucha (si la lucha se gana, ni se gana ni se pierde energía).
- Nodos *obstáculo*. Nodos que representan superación de obstáculos. Son aquellos que no tienen hijo izquierdo, pero sí hijo derecho, que representa la continuación del juego una vez superado el obstáculo. El valor del nodo representa la energía que se pierde al superar dicho obstáculo.
- Nodos *alimento*. Nodos que representan consumo de alimentos. Son aquellos que no tienen hijo derecho, pero sí hijo izquierdo, que representa la continuación del juego una vez consumido el alimento. El valor del nodo representa la energía que se recupera al consumir dicho alimento.

- Nodos *finales*. Nodos que representan el fin del juego. Son los nodos hoja del árbol, y su valor es siempre 0.

El siguiente es un ejemplo de este tipo de árbol:



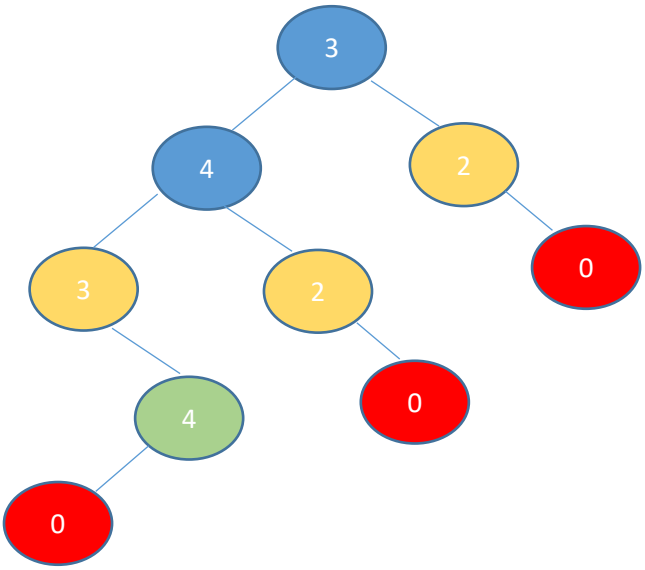
El nodo azul es un nodo *lucha*, el naranja es un nodo *obstáculo*, el verde es un nodo *alimento*, y los rojos son nodos *finales*.

El juego representado por este árbol será jugable para una energía inicial de 5 pues puede haber tanto partidas que se ganan como partidas que se pierden. Efectivamente, si el jugador ganase la lucha (que está en el nodo raíz) llegaría al nodo obstáculo con la energía inicial y tras superar el obstáculo seguiría sin perder la partida (energía = 1) y alcanzaría un nodo final (es decir, podría haber, al menos, una partida ganadora); si, por el contrario, el jugador perdiese la lucha terminaría esa acción sin energía y habría una partida que se pierde.

Ese mismo juego con energías iniciales menores que 5 no sería jugable pues no habría partidas ganadoras. Por ejemplo, con energía inicial igual a 4, si ganase la lucha acabaría sin energía tras superar el obstáculo, con lo que esa partida sería una partida perdedora. La otra posible partida, en la que el jugador perdería la lucha, le haría salir de la lucha con energía menor que 0 y también sería una partida perdedora.

Si ese mismo juego se inicia con una energía mayor que 5 tampoco sería jugable pues no habría partidas perdedoras. Por ejemplo, con energía inicial igual a 6, si ganase la lucha superaría el obstáculo con energía 2 y llegaría al nodo final, es decir, sería una partida ganadora. Si no ganase la lucha, llegaría al nodo final con energía 5, es decir, también sería una partida ganadora.

Este otro juego



no es jugable para un jugador con energía inicial 3 (todas las partidas se pierden) pero sí lo es para un jugador con energía inicial 4 (hay 2 partidas que se pierden y una se gana).

Debes implementar un algoritmo eficiente que, dado un árbol binario que representa un juego de `FunnyGames`, y dada la cantidad inicial de energía del jugador, determine si el juego es o no *jugable* para dicha energía inicial. Debes, así mismo, determinar justificadamente su complejidad.

Ejercicio 3 [5 puntos]

Nos han encargado implementar un sistema para la gestión de los pacientes de un centro de salud. Cada médico del centro tiene un número de colegiado y puede atender un número máximo de pacientes por día. Cada paciente del centro tiene un código de identificación y un nombre. Cada médico tiene vinculados un grupo de pacientes, que son los que,

en caso de necesidad, pueden pedir cita con él para que los atienda. Además, los pacientes pueden cancelar una cita que hayan solicitado previamente, y se deben poder obtener listados tanto de los pacientes citados con un médico como de todos los pacientes vinculados a él.

Para llevar a cabo la implementación de este sistema hemos decidido desarrollar un TAD `CentroSalud` con las siguientes operaciones:

- `centroSalud()`: Operación constructora que crea un sistema de gestión de pacientes vacío.
- `annadir_medico(num_colegiado, num_pacientes)`: Añade un nuevo médico al sistema, con número de colegiado `num_colegiado` y capacidad para atender `num_pacientes` pacientes como máximo al día. Si ya está dado de alta en el sistema un médico con dicho número de colegiado, la operación lanzará una excepción `EMedicoExistente`.
- `annadir_paciente(codigo_id, nombre, num_colegiado)`: Añade un nuevo paciente al sistema, con código de identificación `codigo_id` y nombre `nombre`, que estará vinculado al médico con número de colegiado `num_colegiado`. Si ya está dado de alta un paciente con código de identificación `codigo_id` o no está dado de alta el médico con el número de colegiado `num_colegiado`, la operación lanzará una excepción `EIncorporacionNoAdmitida`.
- `citar(codigo_id)`→`boolean`: Permite al paciente con código de identificación `codigo_id` concertar una cita con el médico al que está vinculado. Si el paciente ya tiene concertada una cita previa con el médico o se ha alcanzado el número máximo de pacientes que pueden citarse con el citado médico en el día, la nueva cita no podrá concertarse y se devolverá `false` no llevándose a cabo ninguna acción; en caso contrario, la cita podrá concertarse (devolverá `true`) y el paciente se colocará en fila detrás del último paciente citado hasta el momento con dicho médico. La operación elevará una excepción `ECitaNoAdmitida` si no existe un paciente con código de identificación `codigo_id`.
- `citados(num_colegiado)->lista<codigo_id>`: Devuelve una lista con los códigos de identificación de los pacientes citados con el médico con número de colegiado `num_colegiado`. En la lista figurará primero el último paciente citado, segundo el penúltimo paciente citado, y así sucesivamente. Si no tiene pacientes citados se devuelve una lista vacía. Si no existe un médico con el número de colegiado dado, la operación elevará una excepción `EMedicoInexistente`.
- `cancelar_cita(codigo_id)`→ `boolean`: Permite cancelar la cita que tenga el paciente con código de identificación `codigo_id`. Si el paciente no tiene concertada ninguna cita, devolverá `false` y no se llevará a cabo ninguna acción; en caso contrario devolverá `true` y el paciente dejará de estar en la fila de paciente citados. La operación elevará una excepción `ECancelacionNoAdmitida` si no existe un paciente con código de identificación `codigo_id`.
- `vinculados(num_colegiado)->lista<codigo_id+nombre>`: Devuelve una lista con los códigos de identificación y nombres de los pacientes vinculados con el médico con número de colegiado `num_colegiado`. La lista estará ordenada crecientemente por código de identificación de los pacientes. Si el médico no tiene pacientes vinculados se devuelve una lista vacía. Si no existe un médico con el número de colegiado dado, la operación elevará una excepción `EMedicoInexistente`.

Ten en cuenta que la implementación de las operaciones debe ser lo más eficiente posible. Por tanto, debes elegir una representación adecuada para el TAD. Además, debes implementar las operaciones y justificar la complejidad de cada una de ellas.