UNIVERSITAT JAUME·I

1º Ingeniería Informática
1º Matemática Computacional
EI1008/MT1008 Programación II
Examen final. Segunda Convocatoria (19 de junio de 2024)

PROBLEMA 1 (4,5 PUNTOS)

Disponemos de una clase, `Ruta`, que permite almacenar la información de una ruta de cicloturismo. La clase proporciona, entre otros, los siguientes constructores y métodos públicos:

- `public Ruta(String nombre, double distancia, int dificultad)`: crea un nuevo objeto de la clase `Ruta` con el nombre de la ruta, su distancia en km y su grado de dificultad (un valor de 0 a 10).
- `public String getNombre()`: devuelve el nombre de la ruta.
- `public double getDistancia()`: devuelve la distancia en km de la ruta.
- `public int getDificultad()`: devuelve el grado de dificultad de la ruta.

Necesitamos implementar una nueva clase, `CatalogoRutas`, que nos permita guardar todas las rutas de una región. Considera que ya tenemos la siguiente definición parcial de la clase:

```
public class CatalogoRutas {
    private Ruta[] vector;  // Vector de las rutas del catálogo
                            // ordenadas de menor a mayor por nombre de ruta
    private static final int TAMAÑO_INICIAL = 10; // Capacidad inicial del vector de rutas
    private int cantidad;   // Cantidad real de rutas en el catálogo


    public CatalogoRutas() {
        this.vector = new Ruta[TAMAÑO_INICIAL];
        this.cantidad = 0;  // Inicialmente el catálogo de rutas está vacío
    }
    ...
```

Añade a la clase `CatalogoRutas` los siguiente métodos públicos:

a) **(1,5 puntos)** `void añadirNuevaRuta(String nombreRuta, double distancia, int dificultad)`

El método añade al catálogo una nueva ruta con el nombre, la distancia y la dificultad dados. Ten en cuenta que las rutas deben quedar ordenadas lexicográficamente por nombre. Además, las rutas se almacenarán siempre en posiciones consecutivas al principio del vector. Cuando sea necesario aumentar la capacidad del vector, deberás duplicar su capacidad. El coste temporal de este método debe ser $\mathcal{O}(n)$.

b) **(1,5 puntos)** `String[] rutasGrado(int dificultad)`

Dado un grado de dificultad, devuelve un vector de cadenas con los nombres de las rutas en el catálogo cuyo grado de dificultad coincide con el dado. El vector devuelto como resultado contendrá los nombres de las rutas ordenados lexicográficamente y no almacenará ningún elemento `null`. En concreto, si en el catálogo de rutas no hay ninguna con el grado de dificultad dado, se devolverá un vector con cero cadenas. El coste temporal de este método debe ser $\mathcal{O}(n)$.

c) **(1,5 puntos)** `CatalogoRutas unir(CatalogoRutas otroCatálogo)`

Este método se comporta de manera que, por ejemplo, la llamada `rutasEspaña.unir(rutasFrancia)` devuelve un nuevo objeto de la clase `CatalogoRutas` que contiene todas las rutas de `rutasEspaña` y todas las rutas de `rutasFrancia`. Puedes asumir que no hay rutas duplicadas, es decir, que si una ruta

está en un catálogo nunca aparecerá en el otro. Debes tener en cuenta que en el catálogo resultante se debe mantener el orden lexicográfico por nombre de ruta y que el coste temporal de este método debe ser $\mathcal{O}(n)$.

Problema 2 (4,5 puntos)

Necesitamos gestionar las solicitudes para realizar visitas guiadas a una serie de monumentos turísticos cuyo acceso es limitado. Para ello, ya disponemos de una clase, `Solicitud`, que proporciona, entre otros, los siguientes constructores y métodos públicos:

- `public Solicitud(String Dni, Fecha f)`: constructor de la clase. A partir del DNI de la persona solicitante y de la fecha[1] de visita deseada, construye la solicitud correspondiente.
- `public String getDni()`: devuelve el DNI de la persona solicitante.
- `public Fecha getFecha()`: devuelve la fecha de visita solicitada.

Además, ya disponemos de la siguiente implementación parcial de la clase `ColaSolicitudes`:

```
public class ColaSolicitudes {

    private static class Nodo {
        Solicitud dato;
        Nodo sig;

        Nodo(Solicitud s, Nodo sig) {
            this.dato = s;
            this.sig = sig;
        }
    }

    // Atributos
    private String monumento;    // Nombre del monumento
    private Nodo primero;        // Enlace al primer nodo de la cola de solicitudes
    private Nodo último;         // Enlace al último nodo

    // Constructor
    public ColaSolicitudes(String monumento) {
        this.monumento = monumento;
        this.primero = null;
        this.último = null;
    }

    ...
}
```

Como puedes ver, la clase `ColaSolicitudes` utiliza nodos con enlace simple para almacenar la información de todas las solicitudes de visita para un monumento. Las solicitudes se mantendrán ordenadas por riguroso orden de llegada, es decir, las nuevas solicitudes se añadirán siempre al final de la cola.

Añade a la clase `ColaSolicitudes` los siguientes métodos públicos:

a) **(0,5 puntos)** `void añadir(Solicitud s)`

El método debe añadir al final de la cola la solicitud `s`. Se exige que su coste temporal sea $\mathcal{O}(1)$.

[1] Considera que tienes disponible la clase `Fecha` desarrollada en prácticas. Recuerda que, entre otros, esta clase proporciona un constructor `Fecha(int día, int mes, int año)` y un método `equals`.

b) **(1,5 puntos)** `void añadir(String nombreFichero) throws FileNotFoundException`

El método debe añadir todas las solicitudes de visita para el monumento en el mismo orden en el que aparecen en el fichero indicado y, lógicamente, debe ignorar las solicitudes de visita para otros monumentos. Este método debe llamar al método `añadir` del apartado anterior cuando sea necesario.

Cada línea de un fichero de solicitudes de visita almacena los datos de una solicitud con el siguiente formato:

```
nombreMonumento DNI día mes año
```

Por simplicidad, los nombres de los monumentos nunca contendrán espacios en blanco. Así, podemos gestionar monumentos como `"AlhambraDeGranada"` o `"GiraldaDeSevilla"`.

Se exige que el coste temporal de este método sea $\mathcal{O}(n)$, donde `n` es la cantidad de líneas en el fichero dado.

c) **(2,5 puntos)** `String[] organizarVisita(Fecha f, int aforo)`

El método permite organizar una visita al monumento en la fecha `f` para un máximo de `aforo` personas. Como resultado, debe devolver un vector de `aforo` cadenas que contenga los DNI de los solicitantes que podrán visitar el monumento el día dado —las últimas componentes del vector contendrán `null` si hubiera menos de `aforo` solicitudes para ese día—.

Las plazas se asignarán según el orden de las solicitudes para la fecha dada. Cuando se haya alcanzado el aforo permitido, para cada una de las solicitudes no aceptadas se deberá llamar al método estático `Notificar.denegar(Solicitud s)`, que notificará al interesado que su solicitud ha sido denegada.

El método también deberá borrar de la cola todas las solicitudes de visita al monumento para la fecha dada, tanto las que han sido aceptadas como las que han sido denegadas.

Se exige que el coste temporal de este método sea $\mathcal{O}(n)$, donde `n` es la cantidad de solicitudes en la cola.