M0.529 – Estructuras de Datos y Algoritmos

PEC 1 – Tipos Abstractos de Datos: Bolsas, Pilas y Colas

Presentación

Esta Prueba de Evaluación Continua (PEC) introduce el concepto central de la asignatura: el Tipo Abstracto de Datos (TAD o ADT en inglés). En particular, trabajaremos una familia concreta de TADs, los contenedores secuenciales, como pueden ser las bolsas, pilas y colas.

La PEC tiene ejercicios de tipo teórico y otros de carácter más práctico, donde será necesario desarrollar fragmentos de código en el lenguaje Java.

Competencias

En esta PEC se desarrollan las siguientes competencias del Máster en Ingeniería Computacional y Matemática:

- Comprender y poder aplicar conocimientos avanzados de computación y métodos numéricos o computacionales a problemas de ingeniería.
- Aplicar los métodos matemáticos y computacionales a la resolución de problemas tecnológicos y de ingeniería de empresa, particularmente en tareas de investigación, desarrollo e innovación.
- Diseñar, implementar y validar algoritmos utilizando las estructuras más convenientes.

Objetivos

Los objetivos concretos de esta PEC son:

- Conocer el concepto de tipo abstracto de datos y saber definir un tipo abstracto de datos para un problema concreto.
- Conocer los tipos abstractos de datos secuenciales (pilas, colas, listas) y sus operaciones, y saber utilizarlos para resolver problemas.

Así pues, deberéis conocer el funcionamiento interno de bolsas, pilas, y colas, así como identificar los pros y las contras de cada una de las estructuras de datos. Además, deberéis ser capaces de analizar problemas y escoger la estructura de datos más adecuada para la resolución del mismo.

Nótese que, salvo que se mencione lo contrario, todas las actividades aquí presentes son obligatorias, y la no entrega de alguna de ellas verá su resultado reflejado en la calificación final de la PEC.

Enunciado

Ejercicio 1 (12,5 %)

Dada la implementación basada en listas enlazadas estudiada en el libro, indica cuál sería el estado de las siguientes estructuras de datos tras realizar las siguientes operaciones, así como los valores devueltos por las operaciones realizadas.

- *Cola*: `enqueue(10), enqueue(2), dequeue(), dequeue(), dequeue(), dequeue(), enqueue(4), enqueue(5), enqueue(3), dequeue()`
- *Pila*: `push(20), push(30), push(10), pop(), pop(), push(45), push(3), pop(), pop(), pop()`
- *Bolsa*: `add(3), add(4), add(14), add(26), add(10)`

Puedes asumir que inicialmente todas las estructuras de datos están vacías.

Ejercicio 2 (25 %)

El objetivo de este ejercicio es el de implementar el *buffer* de un editor de textos. Los métodos que deberá proporcionar vuestro TAD serán los siguientes:

- Crear un *buffer* vacío
- Insertar un carácter en la posición del cursor
- Eliminar y devolver el carácter en la posición del cursor
- Mover el cursor *k* posiciones a la izquierda
- Mover el cursor *k* posiciones a la derecha
- Devolver el número de caracteres que hay en el *buffer*

Se recomienda que utilicéis, para su implementación, dos pilas. Cada una de ellas representará el espacio de *buffer* a la derecha y a la izquierda del cursor.

Encontraréis un esqueleto de la clase a implementar en ***edu.uoc.mecm.eda.pac1.exercise2.TextEditorBuffer***.

Adicionalmente, encontraréis la clase ***edu.uoc.mecm.eda.tests.TextEditorBufferTest***, que contiene una serie de test unitarios que vuestra implementación deberá superar de forma obligatoria.

Ejercicio 3 (25 %)

Para desarrollar aplicaciones cada vez más robustas, la empresa donde trabajamos ha comenzado a explorar el desarrollo de microservicios desplegados en contenedores. Esta es una de las bases de la filosofía *DevOps*, que consiste en descomponer una aplicación compleja en piezas más

pequeñas y manejables, para poder conseguir una serie de ventajas, como pueden ser la escalabilidad horizontal.

En este sentido, cada uno de estos microservicios se podrá desplegar en lo que se llama un ***Contenedor*** (que vendría a ser como una máquina virtual ligera) y es en este entorno donde vivirá, esperando recibir peticiones para resolverlas.

Evidentemente, cada uno de estos contenedores pueden ver degradado su funcionamiento a lo largo del tiempo, o bien pueden llegar a colgarse por algún error de programación o condición excepcional. Así pues, nos hace falta una entidad superior que sea capaz de coordinarlos y pueda, por ejemplo, reiniciar un contenedor que ha caído de forma automática, pueda incrementar su número de réplicas para gestionar la alta demanda, etc. Este tipo de entidad es la que se llama un *orquestador*.

Para que el orquestador pueda manejar los contenedores, este necesita agruparlos en lo que se denominan **Servicios**. Un servicio es una entidad que permite agrupar uno o más contenedores, donde cada uno de ellos sería una instancia de un microservicio que puede ejecutarse en múltiples servidores. Para entenderlo mejor, podemos considerar la definición de un servicio llamado *MySQL*, que consta de dos contenedores (*mysql_0* y *mysql_1*), cada uno de los cuales se ejecuta en un servidor diferente proporcionando alta disponibilidad.

Después de esta contextualización inicial, fijémonos, pues en la especificación de estos Tipos Abstractos de Datos (TADs):

- **CONTAINER**, es la unidad mínima de ejecución dentro de nuestra arquitectura. Deberá almacenar un identificador, su nombre descriptivo, quien es su propietario, la fecha de puesta en marcha y su dirección IP interna (que puede cambiar con el tiempo).

- **SERVICE**, es la entidad que agrupa uno o más contenedores que prestan un mismo servicio. Almacenará su identificador, su nombre descriptivo, su dirección IP externa (no cambiará habitualmente) y su nombre de DNS interno, así como el conjunto de containers que lo forman.

Una vez descrito el funcionamiento esencial, responded las siguientes preguntas:

1. Nuestro sistema debe permitir agregar, para un servicio determinado, el conjunto de contenedores que lo forman. Asimismo, cuando se quiera obtener la lista de contenedores de un servicio, tendremos que obtenerla de forma ordenada por orden de entrada al sistema. ¿Cuál de las estructuras de datos **indicadas en la sección "Recursos" de este enunciado** consideráis que es la más adecuada para almacenar el conjunto de contenedores en el orden en que se nos pide que lo hagamos? Razonad vuestra respuesta.

2. Echad un vistazo a los mecanismos de encapsulación disponibles en el lenguaje Java (https://www.tutorialspoint.com/java/java_encapsulation.htm). A continuación, abrid las clases llamadas ***edu.uoc.mecm.eda.pac1.exercise3.Service*** y ***edu.uoc.mecm.eda.pac1.exercise3.Container*** e implementad los métodos que faltan para que las pruebas contenidas en la clase ***edu.uoc.mecm.eda.tests.ContainerClusterTest*** sean correctas.

3. Definid e implementad en Java un TAD llamado ***Cluster*** que permita contener la información de todos los servicios que se ejecutan dentro de nuestro sistema. El

objetivo debe ser minimizar el espacio ocupado, maximizando también la eficiencia en la medida de lo posible. A priori tendremos que ser capaces de almacenar un número indeterminado de servicios. ¿Qué tipo de estructura de datos de los indicados en la **sección "Recursos" de este enunciado** utilizaríais y por qué? ¿Cuál de las clases disponibles en la **Collections Framework** (https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Collections.html) de Java creéis que es la más adecuada para este propósito? Implementad los siguientes métodos del TAD ***Cluster***:

```
addService (Service s)

removeService (int serviceId)

getClusterSize() -> int

getService (int serviceId) -> Service
```

No es necesario verificar si un servicio está repetido o no al almacenarlo. Tened en cuenta que dentro de la clase ***Cluster*** deberéis incluir una variable privada del tipo de estructura de datos elegida para almacenar la información de los servicios contenidos.

4. El tratamiento de las excepciones es muy importante a la hora de programar y trabajar con TADs. Por lo tanto, debéis controlar que, si intentáis añadir un contenedor a un servicio con la misma IP interna que otro, el sistema deberá lanzar la excepción ***ContainerIPAlreadyBoundException***. Por otro lado, si intentáis obtener un servicio que no existe, tendréis que lanzar una excepción ***NoSuchElementException.*** Estas condiciones se comprueban en las pruebas proporcionadas, por lo que para superarlas satisfactoriamente deberéis tener esto en cuenta en vuestro código.

Para comprobar que la implementación funciona correctamente, debéis aseguraros de que todas las pruebas contenidas en la ***clase edu.uoc.mecm.eda.pac1.exercise3.ContainerClusterTest*** pasen sin ningún error.

Ejercicio 4 (12,5 %)

Se considera una buena práctica del desarrollo de microservicios y aplicaciones en general, que estos dejen constancia de ciertos eventos que pasan a lo largo de su ciclo de vida. Por ejemplo, durante su inicialización, durante el transcurso de su ejecución, si mueren inesperadamente y por qué motivo, etc. Estos eventos se anotan en una especie de diario que se llama ***log***.

Existen muchas formas de almacenar estos *logs*, pero si hay una constante bastante extendida en la mayoría de los sistemas informáticos existentes es que su tamaño no es infinito. En este sentido, estos *logs* normalmente se almacenan en la memoria RAM del sistema informático hasta un tamaño máximo. Cuando se llega a este tamaño máximo, las entradas del *log* se persisten en almacenamiento físico (disco) y a continuación se vuelve al inicio de la estructura de datos y se empieza de nuevo, reemplazando el contenido existente.

Se pide, pues, que utilizando la estructura de datos que consideréis más adecuada **de las indicadas en el apartado "Recursos"** de este enunciado, implementéis una estructura de almacenamiento de *logs* que siga estas directrices:

- Se inicialice con una capacidad máxima especificada
- Disponga de dos apuntadores internos: uno al inicio y otro al final de la estructura de datos
- Permita añadir una entrada al *log*, comprobando que no se haya excedido la capacidad de la estructura. Si ésta se ha excedido, hay que lanzar una ***BufferOverflowException***.
- Permita obtener la entrada del *log* apuntada por el apuntador de inicio eliminándola de la estructura. Si no existen entradas, hay que lanzar una ***EmptyBufferException***.
- Permita obtener la entrada del *log* apuntada por el apuntador de inicio sin borrarla. Habrá que comprobar que previamente existan entradas, como en el caso anterior.
- Permita comprobar si la estructura de datos está vacía
- Devuelva el tamaño de la estructura de datos
- Permita borrar la estructura de datos

Para facilitar vuestra tarea, hemos incluido en el proyecto IDEA de la PAC una clase llamada ***edu.uoc.mecm.eda.pac1.exercise4.Log***, que tendréis que implementar. Además, encontraréis la clase ***edu.uoc.mecm.eda.tests.LogTest*** que contiene un conjunto de test unitarios que deben pasar todos para garantizar la correcta implementación de vuestra clase.

Para que se os evalúe correctamente este ejercicio, la implementación de la clase ***Log*** no deberá utilizar **NINGUNA** de las estructuras que ya proporciona Java dentro de la *Collections framework*.

Ejercicio 5 (25 %)

Escribid un método que reciba como parámetro de entrada una cola de enteros y que los deje ordenados de forma creciente teniendo en cuenta que la cola ya se encuentra ordenada en valor absoluto. Por ejemplo, imaginémonos que tenemos la siguiente secuencia de valores almacenados en la cola:

```
[3, -6, -8, 12, -15, 28, -28, 30, -31, 33]
```

Fijémonos como los valores aparecen ya ordenados en valor absoluto (si ignoráis el signo de los números). La llamada a nuestro método deberá reordenar los valores de forma que la cola almacene la siguiente secuencia de valores:

```
[-31, -28, -15, -8, -6, 3, 12, 28, 30, 33]
```

Como resultado, los valores ahora aparecen ordenados de forma creciente teniendo en cuenta el signo de los números.

Para poder resolver el problema, tendréis que utilizar una **pila** como almacenamiento auxiliar.

Dentro del paquete ***edu.uoc.mecm.eda.pac1.exercise5*** encontraréis la clase ***Rearrange*** que contiene el método `rearrange (Queue q)` que deberéis implementar con vuestra solución.

Igualmente, encontraréis la clase ***edu.uoc.mecm.eda.tests.RearrangeTest***, que contiene un conjunto de métodos para testear vuestra implementación y que deberán pasar todos correctamente de forma obligatoria.

Recursos

Los siguientes recursos son de utilidad en la elaboración de esta PEC:

Básicos

- Secciones del libro "*Algorithms, 4th edition*"
    - 1.1 Introducción a Java (págs. 8 – 53)
    - 1.2 Tipos abstractos de datos (págs. 64-113)
    - 1.3 Bolsas, pilas y colas (págs. 120-159)
- Guía de Estudio de la unidad U1
- Guía de Instalación de Java y del entorno IntelliJ IDEA

Complementarios

- Módulo UOC M1 Tipos abstractos de datos
- Módulo UOC M3 Contenedores secuenciales

Muchas de las actividades planteadas en esta PAC requerirán el uso de colas y pilas. En Java, estas clases ya están implementadas de forma nativa, por lo que no necesitas implementarlas de nuevo. Si es la primera vez que usas estas estructuras en Java, quizás deberías echarle un vistazo a las siguientes páginas de la documentación de Java:

- La clase *Stack* implementa una pila basada en un array / vector creciente de elementos https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Stack.html
- *Queue* es una interfaz común a todas las implementaciones de cola en Java https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Queue.html. Se puede usar *ConcurrentLinkedQueue* para colas con implementación basada en listas enlazadas (https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ConcurrentLinkedQueue.html).

Criterios de evaluación

Los **ejercicios 1 y 4** tienen un peso del **12,5% cada uno**.

Los **ejercicios 2, 3 y 5** tienen un peso del **25% cada uno**.

En los ejercicios se valorará la corrección de la respuesta y los argumentos utilizados en cada apartado. Se valorará también la corrección del software desarrollado (superando los juegos de prueba disponibles en forma de tests unitarios – sin cambiarlos en ningún modo), la correcta elección

de la estructura de datos más apropiada, la justificación de dicha elección y la legibilidad del código entregado.

Entrega

Para la entrega de esta PEC se deberán realizar dos entregas:

- Un documento PDF con las respuestas a los ejercicios 1 y 3.
- Una copia de tu proyecto IDEA de la PEC (en formato ZIP) con la resolución de los ejercicios 2, 3, 4 y 5. Es muy importante que incluyáis el directorio completo con vuestro proyecto IDEA entero. Si no lo está, esta parte quedará como no evaluada.

Es importante que incluyáis vuestro nombre tanto en el documento PDF como en el código entregado.

Estos ficheros deberán entregarse en el **Registro de Evaluación Contínua (REC)** del aula antes de las **23:59 del día límite de entrega**.

Antes de realizar vuestra entrega, por favor aseguraos que:

- **El proyecto IDEA que entregáis esté completo.** En el caso de que se requiera de alguna instrucción adicional para que funcione adecuadamente, podéis incluir un pequeño README indicando los pasos a seguir.

Después de realizar vuestra entrega, por favor aseguraos que:

- Os descargáis las versiones que habéis subido en el Registro de Evaluación Contínua (REC) y comprobáis que efectivamente todo lo que teníais que subir está correctamente subido y no os habéis dejado nada.

NOTA IMPORTANTE: Sólo podéis utilizar las estructuras de datos estudiadas en el apartado "Recursos" de este enunciado para resolver los ejercicios de esta PEC.

Recordad que no se aceptarán entregas fuera del plazo establecido y que, si por equivocación entregáis el contenido incorrecto o incompleto, la valoración de vuestra PEC se realizará conforme a lo entregado.