# Estrategias de Programación y Estructuras de Datos

# Tareas de Evaluación Septiembre 2020

## Normativa general

En este documento se encuentra el enunciado de las cuatro tareas de evaluación para la convocatoria de septiembre de 2020. Junto a este documento se encuentra un directorio con la estructura de clases para realizar las tareas de programación.

El plazo de entrega de las mismas se extiende desde las 10:00 del martes 1 de septiembre hasta las 10:00 del sábado 5 de septiembre.

La respuesta a las tareas se entregará a través de la misma tarea donde se ha publicado este enunciado, en la página de Evaluación 2020 del curso virtual:

https://2020.cursosvirtuales.uned.es/dotlrn/grados/asignaturas/71901043-20/one-community?page_num=6

El formato de entrega será un fichero comprimido en ZIP que deberá contener:

- El directorio con la estructura de clases proporcionado por el Equipo Docente, en el que se incluirán las clases que habrán de ser programadas en las diferentes actividades.
- Cuatro documentos en formato PDF (uno por tarea) con la respuesta a las cuestiones teóricas planteadas en cada una de las tareas. El nombre de cada uno de estos documentos incluirá el número de tarea y los apellidos y nombre del estudiante, de la siguiente manera: "T1 Perez Gomez Ana.pdf", "T3 Martin Gonzalez Pedro.pdf", etc.
- En la primera página de cada uno de estos documentos se deberá indicar claramente nombre, apellidos y DNI/pasaporte del estudiante.

Por restricciones de almacenamiento, **el fichero comprimido no podrá superar los 10MB de tamaño**.

Para la evaluación de las respuestas, además de la **corrección** y **justificación de los razonamientos**, se valorará la **claridad de las explicaciones** y la **presentación** de las mismas.

**Tarea 1: listas implementadas mediante pilas**
Calificación: 2.5 puntos

Se proporciona una clase `StackAsList<E>` totalmente programada en el fichero `StackAsList.java` que implementa una lista con acceso por posición (según la interfaz `ListIF<E>`) utilizando una pila (`StackIF<E>`) como estructura de soporte.

Se pide:

1. Calcule el coste asintótico temporal en el caso peor de los métodos `get`, `set`, `insert` y `remove` de la clase `StackAsList<E>` de acuerdo al código programado. Se deberá justificar el cálculo del coste realizado.

2. Supongamos ahora que queremos que la clase `StackAsList<E>` no sólo implemente `ListIF<E>` sino también `IteratorIF<E>`. Es decir, la declaración de la clase pasaría a ser:

   ```
   public class StackAsList<E> implements ListIF<E>, IteratorIF<E>
   ```

   lo que significa que la clase deberá implementar, además, los métodos `hasNext()`, `getNext()` y `reset()` prescritos por `IteratorIF<E>`, los cuales deberán funcionar **sin modificar el contenido de la pila `stack`**. Entonces:

   a) Teniendo en cuenta **únicamente la definición** del atributo `stack` de la clase, ¿sería posible acceder directamente a la estructura de la pila? Justifique su respuesta.

   b) Describa, no es necesario implementar, cómo implementaría esos tres métodos **sin llamar a ningún método `iterator()` (ni el de la propia clase ni el de `Stack`)**. No olvide indicar qué modificaciones serían necesarias en la estructura de datos para que los métodos funcionen correctamente.

   c) Calcule el coste asintótico temporal, de acuerdo a la respuesta al punto anterior, de los tres métodos declarados en `IteratorIF<E>`. ¿Considera adecuado el uso de estos métodos frente al uso de un iterador de la clase generado por el método `iterator()`? Justifique su respuesta.

**Tarea 2: el problema de la bandera holandesa**
Calificación: 2.5 puntos

El **problema de la bandera holandesa** se trata de un problema de ordenación propuesto por el científico neerlandés Edsger W. Dijkstra. Puede enunciase de la siguiente manera: supongamos una lista no vacía de elementos cuyos únicos posibles valores son 'rojo', 'blanco' y 'azul', de forma que la lista contiene al menos un elemento de cada color. El objetivo consiste en ordenar la lista de manera que en sus primeras posiciones se encuentren los elementos de valor 'rojo', seguidos a continuación por los elementos de valor 'blanco' y, finalmente en las últimas posiciones, los elementos de valor 'azul'.

**Programación:** el ejercicio de programación de esta tarea tiene dos partes:

1. Programe un método `swap(ListIF<E> L, int i, int j)` que intercambie los valores de las posiciones i-ésima y j-ésima de la lista `L`.

2. Programe un método `dutchFlag(ListIF<Character> L)` que resuelva el problema de la bandera holandesa **haciendo uso del método *swap***. Se asume que la lista L cumple las condiciones del problema descritas anteriormente. Los valores de la lista se representarán del siguiente modo: R (rojo), B (blanco) y A (azul).

Se proporciona una clase `DutchFlag.java` con el esqueleto de los métodos pedidos y un método `main` que realiza una prueba del método `dutchFlag` usando un método que genera de forma pseudoaleatoria una lista de caracteres de longitud mínima `n` compuesta de valores '`R`'(rojo), '`B`' (blanco) y '`A`' (azul) y otro método que muestra el contenido de una lista dada como entrada. Recomendamos que los estudiantes realicen más pruebas a fin de comprobar que su implementación cumple los requisitos indicados.

**Aclaraciones:**
- **Los estudiantes deben entregar solamente el fichero de la clase `DutchFlag.java` con los métodos pedidos programados. No se tendrán en cuenta otros ficheros.**
- **No se permite el uso de iteradores para resolver el problema.**
- **No se permite modificar el esqueleto de los métodos pedidos ni programar métodos auxiliares.**
- **Se valorará la eficiencia de la solución, la claridad del código y los comentarios que expliquen la solución del estudiante.**

**Parte teórica:**

**1.** Calcule el coste temporal en el caso peor del método `dutchFlag`.
**2.** Dados los métodos **F y G** que pueden encontrarse en la siguiente página, se pide:

a) Explique qué calculan ambos métodos. ¿Qué propiedad cumple la lista dada como entrada en ambos métodos en el caso peor en términos de coste computacional?
b) Escriba una recurrencia que exprese su función de coste en el caso peor para cada uno de ellos.
c) Obtenga el coste en el caso peor de ambos métodos a partir de las recurrencias anteriores empleando las ecuaciones de resolución de recurrencias proporcionadas por el Equipo Docente.

```java
public static <E extends Comparable<E>> void F(ListIF<E> L){
	Faux(L,L.size());
}

public static <E extends Comparable<E>> void Faux(ListIF<E> L,int s){
	if(s<=1){
		return;
	}
	for(int i=1;i<s;i++){
		if(L.get(i).compareTo(L.get(i+1))>0){
			swap(L,i,i+1);
		}
	}
	Faux(L,s-1);
}


public static <E extends Comparable<E>> void G(ListIF<E> L){
	Gaux(L,L.size());
}

public static <E extends Comparable<E>> void Gaux(ListIF<E> L,int s){
	if(s<=1){
		return;
	}
	for(int i=1;i<s;i++){
		if(L.get(i).compareTo(L.get(i+1))>0){
			swap(L,i,i+1);
			Gaux(L,s-1);
		}
	}
}
```

**Tarea 3: enriquecimiento de las operaciones de las colas**
Calificación: 2.5 puntos.

Una cola es una estructura de datos que permite el acceso a sus elementos siguiendo el criterio FIFO (del inglés First In, First Out, “primero en entrar, primero en salir”). Sin embargo, en ciertas ocasiones, puede ser interesante que un elemento pueda saltar a la primera posición, o que el último elemento pueda dejar la cola antes.

**Programación**: implemente un código en Java (no es necesario implementarlo en un fichero de desarrollo `.java`) con los siguientes métodos:

```
void jumpQueue(E elem);
void leaveQueue();
```

que permitan, respectivamente, insertar un elemento en el principio de la cola y sacar el elemento que se encuentra en el final de una cola no vacía. Para la realización de estos métodos, **sólo se pueden utilizar las operaciones de colas disponibles en el TAD QueueIF<E> proporcionado por el Equipo Docente** (es decir, las operaciones han de ser válidas para cualquier implementación de dicho TAD) y, además, **no se permite el uso de ninguna estructura de datos auxiliar**. Defina las precondiciones de ambos métodos y justifique cada decisión tomada en el código.

**Parte teórica**:

1. Calcule y justifique el coste asintótico temporal en el caso peor de las dos operaciones programadas (`jumpQueue` y `leaveQueue`) con las restricciones impuestas.

2. La mezcla de varias colas es una operación muy utilizada en la vida real. Imaginemos, por ejemplo, la cola de un hospital en la que hay dos ventanillas y una se cierra. En estos casos hay muchas formas de mezclar ambas colas. En este ejercicio se pide describir, no es necesario implementar, un algoritmo para realizar el método:

   ```
   QueueIF<E> mix(QueueIF<E> q1, QueueIF<E> q2);
   ```

   que mezcla todos los elementos de dos colas dadas por parámetro intercalando (mientras haya elementos en ambas colas) en orden los elementos de `q1` y `q2` comenzando por el primer elemento de la cola `q1`.

   Cada decisión tomada debe estar debidamente justificada. Calcule y justifique el coste asintótico temporal en el caso peor del método `mix`. Se valorará la eficiencia de la solución propuesta.

**Tarea 4: árboles binarios de búsqueda y árboles AVL**
Calificación: 2.5 puntos.

**Parte teórica**:

1. En la asignatura hemos trabajado con árboles binarios de búsqueda (ABB) que organizan conjuntos de elementos sin elementos repetidos. A menudo, sin embargo, nos encontramos problemas en los que los datos pueden estar repetidos. Considera estas variaciones de un ABB para incorporar elementos repetidos:

   - **ABB-1**: Todos los elementos del subárbol izquierdo son **menores o iguales** que la raiz, y todos los elementos del subárbol derecho son mayores que la raiz.
   - **ABB-2**: Todos los elementos del subárbol izquierdo son **menores o iguales** que la raiz, y todos los elementos del subárbol derecho son **mayores o iguales** que la raiz.
   - **ABB-3**: Todos los elementos del subárbol izquierdo son menores que la raiz, todos los elementos del subárbol derecho son mayores que la raiz, y en la raiz incorporamos **un parámetro que indica el número de repeticiones** del valor almacenado en ella.

   Responda razonadamente a las siguientes preguntas:

   a) Para cada variante, ¿cuál es el coste asintótico temporal en caso peor de buscar todas las repeticiones de un elemento?
   b) ¿Cuál de las variantes requiere menor número de operaciones y por qué?
   c) Para cada variante, ¿entraría en conflicto con las rotaciones para mantener la condición de árbol AVL?

2. Considérense tuplas `(a,b)` en las que a es de tipo A y b es de tipo B, y ambos tipos tienen un orden total (es decir: para cualesquiera a y a' en uno de esos tipos, si a ≤ a' y a' ≤ a, entonces a = a'). Definimos el siguiente orden parcial (en el que no se cumple la propiedad anterior) entre tuplas:

   ```
   (a,b) > (a',b') si y sólo si a > a' y b > b'
   (a,b) < (a',b') si y sólo si a < a' y b < b'
   ```

   **Nota**: En un orden parcial no todos los pares de elementos tienen una relación de orden. En el caso que nos ocupa, si los dos tipos A y B son enteros, tendríamos como relaciones de orden `(3,9) < (5,15)` o `(20,30) > (13,2)`. Pero no habría relación de orden entre `(30,5)` y `(12,18)`, por ejemplo.

3. Responda razonadamente a las siguientes preguntas:

   a) ¿Se puede utilizar la estructura de árbol binario de búsqueda para almacenar tuplas `(a,b)` utilizando este orden parcial?
   b) Si nuestro propósito al utilizar un ABB es hacer eficiente la búsqueda de tuplas, ¿cómo utilizaría los ABB en este caso, sin cambiar su interfaz ni su implementación?
   c) Imagine una adaptación de los ABB que los convierta en árboles ternarios, en los que el subárbol adicional (el central) almacene los elementos que no son ni mayores ni menores que la raíz. ¿Qué ventajas e inconvenientes tendría respecto a la solución de la pregunta anterior?