Estructuras de datos y algoritmos

Ejercicios de la primera parte: Análisis. Pilas. Colas. Listas.

Todos los algoritmos que se escriban deben ser analizados, y la justificación del análisis debe presentarse junto con el algoritmo.

.

Los marcados con (Tipo 0) son obligatorios. El número que acompaña a la marca de Tipo sugiere la dificultad del ejercicio. Cuanto mayor es el número, mayor dificultad esperada.

1. (Tipo 0) Implementar OrdenaPorSelección.

2. (Tipo 0) Implementar OrdenaPorInserción.

3. (Tipo 0) Implementa la clase StackOfT<T> para representar pilas genéricas, usando enlaces. Debe colocarse en el paquete ehu.eda.list.linkedImp del proyecto java que vamos desarrollando en clase.

4. (Tipo 1) Implementar una clase para manipular pilas de cajas. Cada caja tiene una talla y número de serie. Las pilas de cajas están sujetas a una restricción: todas los cajas de la misma talla están apiladas juntas. Completar la implementación de la clase, que se presenta a continuación, para que responda a los requisitos indicados:

```
public class PilaDeCajas {
        //Definir variables de instancia

    public void addCaja(int talla, int id);
    Postcondición: añade la caja a la pila.
    Se añade junto a las cajas de su talla, si es que las hay.
    En caso contrario, se añade en la cima de la pila.

    public int rmv();
    Precondición: la pila no está vacía.
    Postcondición: quita la caja de la cima de la pila
    y devuelve su número de serie.

    public int rmv(int talla);
    Precondición: la pila no está vacía.
    Postcondición: retira de la pila todas las cajas
    con la talla indicada. Devuelve el número de cajas retiradas.
}
```

5. (Tipo 1) Completar la implementación de la clase TransformaApp (que está en el paquete aplicacionesDePilas). Se trata de implementar dos métodos:

```
/* Devuelve otra pila, con los números pares de p situados
* en el mismo lugar que estaban en p y, sin embargo,
* los números impares ocuparán los mismos lugares pero
* estarán en orden inverso al que estaban en p.
* Ejemplo: Si p=[2, 4, 3, 5, 6, 7, 9] (siendo 9 la cima),
```

```
* se devolverá la pila [2, 4, 9, 7, 6, 5, 3]
*/
private static StackOfT<Integer> Transforma(StackOfT<Integer> p) {
                // HACER
}

//Escribe los elementos de la pila p desde el fondo a la cima
private static void Imprime(StackOfT<Integer> p) {
                // HACER
}
```

6. (Tipo 2) En la cola de una oficina virtual se acumulan *solicitudes* enviadas a gestionar por la oficina. Cada solicitud tiene asignada una prioridad, que es un entero entre 1 (mínima prioridad) y 9 (máxima prioridad). La elección de la siguiente solicitud a gestionar, es como sigue:

   - se retira la primera solicitud de la cola (llamémosle X).
   - si hay en la cola alguna solicitud con prioridad mayor que la de X entonces se añade X al final de la cola y no se gestiona X.
   - si no hay tal solicitud, se gestiona X (que, naturalmente, no se vuelve a incorporar a la cola).

   Imagina que se sabe cuántas solicitudes hay en la cola, cuál es la prioridad de cada solicitud y cuál es la posición de tu solicitud en la cola.

   El problema consiste en calcular cuántas solicitudes se gestionarán antes que la tuya, supuesto que no se añaden nuevas solicitudes a la cola.

   Para resolverlo se incluirá un programa que procese un archivo de entrada con el siguiente contenido:

   - Primera línea: dos enteros N y P separados por (uno o más) espacios en blanco. N es el número de solicitudes en la cola. P es la posición de tu solicitud en la cola, cumpliéndose siempre que: $1 \leq P \leq N$
   - Línea siguiente: N enteros en el rango [1..9], y separados entre sí por uno o más espacios en blanco. Esos N números son las prioridades de las N solicitudes contenidas en la cola.

   Así, el contenido del archivo de entrada podría ser:

   ```
   6    1
   1   1   9   1   1   1
   ```

   Y la salida debería ser:

   ```
   4
   ```

   En este problema es importante analizar el tiempo de ejecución del programa. La solución será valorada según la eficiencia del programa.

7. (Tipo 2) Una cola *multiusuario* es una cola usada por un número determinado de usuarios para intercambiar mensajes. Los usuarios se suponen numerados consecutivamente y hay una cantidad fija N de usuarios.

   El uso de una cola multiusuario sigue las reglas siguientes:

- Cualquier usuario puede añadir mensajes (al final de la cola). Usando el método `incluye(String msg)`.
- Cualquier usuario puede leer mensajes. Cada usuario lee los mensajes en el mismo orden en que fueron añadidos a la cola. Cuando un usuario lee un mensaje, ese mensaje es el que más tiempo lleva en la cola de los que le faltan por leer a ese usuario, y ese mensaje queda leído por ese usuario. Cuando un usuario lee un mensaje, éste no desaparece de la cola todavía. Se usa el método `leer(int usuario)`.
- Cualquier usuario puede purgar la cola: se retiran de la cola los mensajes que han sido leídos por todos los usuarios. Usando el método `purgar()`.

Se pide completar la clase siguiente para manipular colas multiusuario:

```
package student;

public class ColaMultiUsuario {
        private final int N; // número de usuarios
        // Definir estructura para representar el estado de la cola

        public ColaMultiUsuario(int usuarios) {
                assert usuarios >= 0; // Precondicion
                N = usuarios;
        }

        public void incluye(String msg) {}

        public boolean hayMensajesPara(int usuario) {
                assert 0 <= usuario && usuario < N // Precondicion
        }

        public void purgar() {}

        public String leer(int usuario) {
                assert 0 <= usuario && usuario < N; // Precondicion
        }
}
```

8. (Tipo 1) Una *DoubleEndedQueue* es una cola particular en la que se puede añadir y retirar elementos tanto por el principio como por el final.

   Completa la implementación de la clase DoubleEndedQueue con los métodos que aparecen a continuación. Debes usar objetos enlazados para representar la cola.

```
public class DoubleEndedQueue<E> {
        // E es el parámetro de clase
        // define las variables de instancia

        //Añade al principio
        public void addFirst(E e) { }

        //Añade al final
        public void addLast(E e) { }

        //Devuelve el primero, sin retirarlo
        public E getFirst() { }
```

```
    //Devuelve el último, sin retirarlo
    public E getLast() { }

    //Devuelve True si la cola está vacía. False si no.
    public boolean isEmpty() { }

    //Devuelve el primero, retirándolo de la cola
    public E removeFirst() { }

    //Devuelve el último, retirándolo de la cola
    public E removeLast() { }
}
```

9. (Tipo 1) Una *DoubleEndedQueue* es una cola particular en la que se puede añadir y retirar elementos tanto por el principio como por el final.

   Completa la implementación de la clase DoubleEndedQueue con los métodos que aparecen en el ejercicio anterior. Debes usar un array para representar la cola.

10. (Tipo 1) Implementa las variantes indicadas del método siguiente:

    ```
    public static int[] RetiraRepetidos ( int[] A ){  }
    ```

    Es importante el análisis de cada variante.

    a) Devuelve un array con los elementos de A, sin que aparezcan elementos repetidos en el array devuelto.

    b) Resuelve el mismo problema, en tiempo $O(n)$, siendo n el número de elementos de A, asumiendo, como precondición, que todos los elementos de A están en el rango [0..K], siendo K mucho menor que el número de elementos de A.

11. (Tipo 2) Implementa el algoritmo de *OrdenaPorRaiz*, que ordena un array de enteros no negativos según el método explicado en clase.

    - (Puntuación extra Tipo 3) Implementa otro algoritmo *OrdenaPorRaizDesc*, adaptando el anterior, para que ordene de mayor a menor. No se aceptan soluciones que consistan en invertir un vector ordenado de menor a mayor.
    - (Puntuación extra Tipo 3) Adapta el algoritmo *OrdenaPorRaiz* para que ordene un array de enteros cualesquiera (positivos y negativos).

12. (Tipo 2) Implementa el método `addAll(LinkedOrdList<T> mset)` de `LinkedOrdList`, vista en clase, de manera que si `this.size()`$=m$ y `mset.size()`$=n$, la función de coste sea de $O(m+n)$.

13. (Tipo 2) Implementa un método `LinkedOrdList<T> intersect(LinkedOrdList<T> mset)` en la clase `LinkedOrdList`, vista en clase, que devuelva una lista ordenada con los elementos que están en `this` y en `mset`. Recuerda que `this` y `mset` son multiconjuntos y, por lo tanto, su intersección puede tener elementos repetidos. El algoritmo debe estar implementado de manera que si `this.size()`$=m$ y `mset.size()`$=n$, la función de coste sea de $O(m+n)$.

14. (Tipo 2) La clase `LinkedMultiSet<T>`, vista en clase, permite crear y manejar multiconjuntos de objetos. Este ejercicio consiste en que implementes otra clase, `Multisetof1000Int`, que permita crear y manejar multiconjuntos de números enteros comprendidos entre 0 y 1000, con las mismas operaciones que `LinkedMultiSet<T>`.

Este ejercicio sólo tiene valor si la función de coste de cada método es independiente del número de elementos que haya en el multiconjunto.

15. (Tipo 3) Imagina que debes gestionar una agenda de tareas, distinguiendo entre tareas *ordinarias* y tareas *prioritarias*. Se quiere tener la posibilidad de añadir tareas a la agenda. También debe ser posible retirar: o bien la tarea que lleva más tiempo en la agenda (sea prioritaria o no), o bien la tarea prioritaria que lleva más tiempo en la agenda. Debes implementar la clase `AgendaDeTareas` de manera que las operaciones marcadas con $O(1)$ se realicen en tiempo constante.

```
public class AgendaDeTareas {
//Las instancias de esta clase representan agendas de tareas.
//Las tareas se identifican por su nombre.

public void addOrdinary(String task); //IMPORTANTE: O(1)
//Añade la tarea a la agenda, considerando esa tarea como ordinaria.

public void addPrioritary(String task); //IMPORTANTE: O(1)
//Añade la tarea a la agenda, considerando esa tarea como prioritaria.

public String rmvPrioritary(); //IMPORTANTE: O(1)
//Precondición: hay tareas prioritarias en la agenda.
//Devuelve, y retira de la agenda, el nombre de la tarea prioritaria
//que más tiempo lleva en la agenda, de todas las prioritarias.


public String rmv(); //IMPORTANTE: O(1)
//Precondición: hay tareas en la agenda.
//Devuelve, y retira de la agenda, el nombre de la tarea
//que más tiempo lleva en la agenda (sea prioritaria o no).

public void rmvNonPrioritary();
//Retira de la agenda todas las tareas no prioritarias.
}
```

Ejemplo de uso:

```
AgendaDeTareas miAgenda = new AgendaDeTareas(); // miAgenda está vacía
miAgenda.addOrdinary("Ordinaria-1");
miAgenda.addPrioritary("Prioritaria-1");
miAgenda.addPrioritary("Prioritaria-2");
miAgenda.addOrdinary("Ordinaria-2");
miAgenda.addPrioritary("Prioritaria-3");
System.out.println(miAgenda.rmvPrioritary()); // escribe Prioritaria-1
System.out.println(miAgenda.rmv()); // escribe Ordinaria-1
System.out.println(miAgenda.rmv()); // escribe Prioritaria-2
```

16. (Tipo 2) Implementa el método constructor de la clase siguiente:

```
import LinkedListOfInt;
public class ParDeListasOrd {
        LinkedListOfInt pares;
        //sólo contiene números pares ordenados de menor a mayor.

        LinkedListOfInt impares;
        //sólo contiene números impares ordenados de menor a mayor.

        public ParDeListasOrd(LinkedListOfInt listaInicial){
```

```
            //HACER
        }
        ......
}

public class LinkedListOfInt {
        Nodo first;

        public static class Nodo{
                int item;
                Nodo next;
                ......
        }
        .....
}
```

El método toma una lista ligada simple (de `LinkedListOfInt`), de valores enteros positivos, y *traslada* sus nodos a dos listas ligadas ordenadas, `pares` e `impares`, que quedarán formadas por los mismos nodos de la lista `listaInicial` dada como parámetro (los números pares quedarán en la primera y los impares en la segunda). Es decir, no deben crearse nuevos nodos sino "trasladarlos"de lugar. Al final, la lista `listaInicial` quedará sin nodos.

17. (Tipo 2) Implementa una clase `Polinomio` para representar polinomios en una variable. Debes hacerlo con una estructura enlazada que use la clase

```
private static class Nodo<T>{
                T item;
                Nodo next;
                ......
}
```

La clase debe incluir estos dos métodos:

```
public Polinomio  suma (Polinomio p)
// suma a this el polinomio p.

public Polinomio  multiplicaPor (Polinomio p)
// multiplica a this por el polinomio p.
```

18. (Tipo 1) Una *Cola de Prioridad* es una colección a la que se añaden elementos con una prioridad asociada (supongamos que representada mediante un número natural) y en la que los elementos se retiran por orden de prioridad (cuanto mayor es el número mayor es la prioridad). Implementa una clase ColaPrioridad<T> con los siguientes métodos: insert(T elem, int p) (que añade un elemento elem con prioridad asociada p a la colección), remove() (que elimina y devuelve el elemento con mayor prioridad asociada), isEmpty() (que devuelve true si la colección no contiene ningún elemento, y false en caso contrario), size() (que devuelve el número de elementos de la cola de prioridad).

19. (Tipo 1) Implementa una clase `Pila50`<T>, que se comporte como una pila con capacidad acotada a 50 elementos. Cuando la pila está llena, se puede seguir insertando elementos, pero por cada elemento que se inserta desaparece de la pila el elemento que más tiempo lleva en la pila.

    Analiza cada método de la clase `Pila50`<T>. La valoración de la implementación puede disminuir si la complejidad es indebidamente ineficiente.

20. (Tipo 1) Cuando el abanico de posibilidades para representar una prioridad está prefijado (y no es muy grande), por ejemplo: sólo pueden ser números enteros del 0 al 9, una representación interesante para una cola de prioridad puede ser un array de colas indexado justamente por ese rango de números. Implementa una clase cola de prioridad utilizando esta idea y discute cuales son las ventajas e inconvenientes de esta implementación frente a las realizadas en los ejercicios anteriores.

21. (Tipo 1) Implementa dos pilas en un array de manera que ninguna de las dos pilas se considere llena, a menos que el array esté lleno. Las operaciones pop() y push() deben ser $O(1)$.

22. (Tipo 0) Implementa y añade los métodos

```
public E get (int index)
public E set (int index, E element)
public void add ((int index, E element)
public E remove (int index)
```

a la clase `IterableDoubleLinkedList<E>`. Recuerda que el `index` del primer elemento es 0 y el resto recibe el index consecutivo que le corresponde.

23. (Tipo 2) Implementa, en la clase `ToyHashMap<K, V>`, el método `public V remove(K key)`.

24. (Tipo 2) Implementa, en la clase `ToyHashMap<K, V>`, el método `public Object[] getAllKeys()`.

25. (Tipo 3) Implementa, en la clase `ToyHashMap<K, V>`, un método `public Object[] getAllKeysInOrder()`, que devuelva un array con las claves de los pares que hay en la aplicación, en el orden en que fueron añadidos a la aplicación.

26. (Tipo 2) Implementa una clase ListaCompartida<T> que represente una lista de objetos (de clase genérica T) compartida por un número de procesos, que se determina como parámetro en la constructora de la clase. Un proceso puede representarse con un número natural.

    La característica principal de esa lista es que cada proceso puede tener acceso a un elemento diferente de la lista en un momento determinado, dependiendo de las operaciones que se hayan realizado sobre la lista hasta ese momento.

    Las operaciones que pueden realizarse sobre una de estas listas son las siguientes:

    - La constructora ListaCompartida (int numProcesos) debe crear una lista para ser compartida por tantos procesos como diga numProcesos, que será mayor o igual que 1.
    - void inserta(T elem): inserta un objeto elem al final de la lista. Puede ser realizada por cualquier proceso.
    - T primeroPara(int p): devuelve el objeto accesible de la lista para el proceso p, sin retirarlo de la lista.
    - void avanza(int p): el objeto accesible de la lista, para el proceso p, pasa a ser el siguiente al que hasta ahora era su objeto accesible en la lista (sólo avanza el proceso p, los demás siguen teniendo el mismo objeto accesible que antes de la operación).
    - T elimina(int p): el proceso p retira de la lista su objeto accesible, y por tanto ese objeto deja de estar en la lista y es inaccesible para los procesos.

- boolean esVacia(int p): devuelve true si el proceso p no tiene objeto accesible en la lista y false en caso contrario.

Así, por ejemplo, imaginemos que se ha construido una lista con los objetos [*x, y, z*] y todos los procesos tienen a x como elemento accesible. Si se realiza la operación avanza(p_1) entonces el objeto y pasa a ser el accesible para p_1 y el resto sigue teniendo como accesible el objeto x. Si, a continuación, se realiza elimina(p_1) entonces la lista pasa a tener los objetos [*x, z*] y el objeto accesible a p_1 es z. En este caso, no se modifican los objetos accesibles para los otros procesos. Si, a continuación, se realiza elimina(p_2) entonces la lista pasa a tener únicamente el objeto [z] y además se habrá modificado el objeto accesible para los procesos que tenían a x como objeto accesible, ahora será el objeto z.

27. (Tipo 1) Añade a la clase `IterableDoubleLinkedList<E>` el método siguiente:

```
//Precondición: la lista está ordenada de menor a mayor.
public void addInOrder(E e)
```

Debes asumir que, si hay elementos en la lista, están ordenados de menor a mayor. `addInOrder(e)` debe añadir el elemento `e` en el lugar de orden que le corresponde. Si no hay elementos, ese elemento `e` será el único de la lista.

28. (Tipo 3) Implementa una clase

```
\texttt{private class IteradorDeArray implements Iterator<Item>}
```

dentro de la clase `ResizingArrayQueue<Item>`. Basta con que implementes los métodos `hasNext()` y `next()`. Añade a la clase `ResizingArrayQueue<Item>` un método que devuelva un objeto `IteradorDeArray`. Utiliza eso, en un programa cliente de `ResizingArrayQueue<Item>`, para construir una cola de palabras (`String`) y comprobar si una determinada palabra está en la cola.