P1. **Pregunta sobre la práctica**. Se desea añadir un nuevo método:

```
public void decFreqQuery (String q);
```

al interfaz `QueryDepotIF`, cuyo cometido es decrementar en uno la frecuencia de una query que ya esté en el depósito de consultas, teniendo en cuenta que se deberá eliminar la query del depósito si la frecuencia llegase a ser 0.

a) (1 Punto) Describa (no es necesario programar) cuál sería el funcionamiento del método para la representación mediante listas de queries ordenadas alfabéticamente.

   En primer lugar deberíamos recorrer la lista hasta encontrar el objeto `Query` correspondiente a la cadena de caracteres `q` que recibimos como parámetro, que sabemos estará por las condiciones del enunciado (de hecho, eso deberá constar en la precondición del método).

   Una vez localizado el objeto adecuado, deberemos decrementar en uno su frecuencia. Si tras decrementar su frecuencia resultase ser cero, tendremos que eliminar este elemento de la lista. Si no lo fuese, no tendríamos que hacer nada.

b) (1 Punto) Describa (no es necesario programar) cuál sería el funcionamiento del método para la representación mediante un árbol de caracteres.

   Al igual que en el apartado anterior, deberemos ir recorriendo el árbol avanzando en cada nodo por el hijo correspondiente al siguiente carácter de la cadena de caracteres `q` que recibimos como parámetro.

   Una vez hayamos llegado al nodo conteniendo el último carácter de la cadena, obtendremos su nodo hijo que contenga la frecuencia de la query (que sabemos ha de existir, ya que la query está en el depósito). De igual forma que en el anterior apartado, habrá que decrementar la frecuencia almacenada en dicho nodo y comprobar si es cero.

   En el caso de que así sea, tendremos que proceder a eliminar el nodo con la frecuencia de la query, ya que ahora esa query desaparece del depósito de consultas. Sin embargo eso no basta, ya que hemos de comprobar si la query era, a su vez, prefijo de otras.

   Para ello, vamos al nodo que contiene el último carácter de la query y comprobamos si tras la eliminación del nodo conteniendo la frecuencia se ha convertido en un nodo hoja. En cuyo caso, subimos a su padre y eliminamos el nodo hoja, repitiendo el proceso hasta que lleguemos a un nodo que tenga más de un hijo, momento en el cual habremos terminado la eliminación.

1. Se dispone de dos programas P_1 y P_2 que resuelven el mismo problema. El tiempo de ejecución de cada uno de ellos en función del tamaño del problema n (respectivamente $T_1(n)$ y $T_2(n)$) viene dado por las siguientes ecuaciones:

$$T_1(n) = \begin{cases} k_1 & si\ n \leq 10000 \\ k_2 \cdot n & si\ n > 10000 \end{cases}$$

$$T_2(n) = k_3 + k_4 \cdot n$$

que nos devuelven las unidades de tiempo que emplea cada programa en ejecutarse en función de n. Asumimos que los valores k_1, k_2, k_3 y k_4 son constantes y conocidos.

a) (1 Punto) ¿Cuál es el coste asintótico temporal en el caso peor de ambos programas? Según el análisis, ¿cuál es asintóticamente más eficiente?

En el caso del programa P_1, vemos que el coste a partir de un tamaño de 10001, es lineal con respecto a dicho tamaño del problema, por lo que el coste asintótico temporal en el caso peor de P_1 será lineal.

En el caso de P_2, el coste es siempre lineal con respecto al tamaño del problema n, con independencia de dicho tamaño.

Así pues, ambos programas tienen un coste asintótico temporal en el caso peor que es lineal con respecto al tamaño del problema, por lo que ninguno es asintóticamente más eficiente que el otro.

b) (1 Punto) ¿Cuál de los dos programas tarda menos en ejecutarse según el valor de n? ¿Cómo podrían combinarse ambos programas para minimizar el tiempo de ejecución para cualquier valor de n?

El tiempo de ejecución depende de los valores concretos de k_1, k_2, k_3 y k_4, por lo que procede una discusión con respecto a los valores relativos de dichas constantes.

- Para tamaños de $n \leq 10000$, P_1 tiene un coste constante k_1, mientras que P_2 tiene un coste de $k_3 + k_4 \cdot n$. Así pues, si $n \leq 10000$, P_1 tarda menos (o lo mismo) en ejecutarse que P_2 si y sólo si $k_1 \leq k_3 + k_4 \cdot n$.
- Para tamaños de $n > 10000$, P_1 tiene un coste $k_2 \cdot n$, mientras que P_2 tiene un coste de $k_3 + k_4 \cdot n$. Así pues, si $n > 10000$, P_1 tarda menos (o lo mismo) en ejecutarse que P_2 si y sólo si $k_2 \cdot n \leq k_3 + k_4 \cdot n$.

Por lo tanto, para un tamaño de problema n y conociendo los valores de k1, k2, k3 y k4, podríamos realizar el siguiente programa (escrito en pseudocódigo):

```
Si n ≤ 10000 {
   Si k1 ≤ k3 + k4·n → ejecutar P1
   Si no → ejecutar P2
}
Si no {
   Si k2·n ≤ k3 + k4·n → ejecutar P1
   Si no → ejecutar P2
}
```

Como las operaciones aritméticas involucradas y las comparaciones pueden realizarse en tiempo constante, ese programa combinaría los programas P1 y P2 de manera que se minimiza el tiempo de ejecución para cualquier valor de n.

2. Se desea disponer de un método:

```
public void removeNonMultiplesQ(int k, QueueIF<Integer> q);
```

que modifique la cola de entrada `q` eliminando de ella todos los elementos que no sean divisibles por `k`, pero manteniendo el orden de los mismos.

a) (1 Punto) Programe en Java el método `removeNonMultiplesQ` descrito anteriormente.

Antes de ver el código vamos a describir el funcionamiento del algoritmo y luego presentaremos el código completo. Tenemos que recorrer todos los elementos de la cola para ir eliminando los que no son múltiplos de `k`. Como eso involucra una modificación, el recorrido no podrá ser realizado con un iterador, así que realizaremos el recorrido empleando las operaciones propias de la cola.

Por lo tanto, obtendremos el primer elemento de la cola y lo eliminaremos. Luego comprobaremos si dicho elemento es múltiplo de `k`, en cuyo caso lo volveremos a introducir en la cola. ¿Cuántas veces deberemos repetir este proceso? Pues tantas como elementos `n` hubiera en la cola al principio.

```
public void removeNonMultiplesQ(int k,
                                QueueIF<Integer> q) {
     int n = q.size();
     for ( int i = 1 ; i <= n ; i++ ) {
          int first = q.getFirst();
          q.dequeue();
          if ( first % k == 0 ) {
               q.enqueue(first);
          }
     }
}
```

b) (1 Punto) Programe en Java el método:

```
public void removeNonMultiplesS(int k, StackIF<Integer> s);
```

que realice la misma operación sobre una pila de enteros.

Realizaremos lo mismo que en el apartado anterior: primero describiremos el funcionamiento del algoritmo y luego presentaremos el código completo.

En primer lugar hay que recordar que una pila sigue una política de gestión LIFO, por lo que como tenemos que conservar el orden de los elementos, es necesario invertir la pila. Para ello, creamos una pila auxiliar donde almacenamos los elementos de la pila original en orden inverso. Aprovechamos este primer recorrido para insertar en la pila auxiliar únicamente aquellos elementos que son múltiplos de `k`.

Una vez hecho esto, tenemos en la pila auxiliar todos los elementos múltiplos de `k` que había en la pila original, pero en orden inverso. Por ello, sólo necesitamos invertir la pila auxiliar en la pila original y ya tendríamos el resultado deseado.

El código es el siguiente:

```
public void removeNonMultiplesS(int k,
                                 StackIF<Integer> s) {
     StackIF<Integer> sAux = new Stack<Integer>();
     while ( !s.isEmpty() ) {
          int top = s.getTop();
          s.pop();
          if ( top % k == 0 ) {
               sAux.push(top);
          }
     }
     while ( !sAux.isEmpty() ) {
          s.push(sAux.getTop());
          sAux.pop();
     }
}
```

Se podría hacer al revés: invertir toda la pila en el primer recorrido y delegar el filtrado de los múltiplos de `k` al segundo recorrido. Sin embargo, esta opción (aunque válida a efectos del examen) es menos eficiente, ya que implica un mayor número de operaciones. Dejamos como ejercicio la comparación del coste (real, no asintótico) de ambas aproximaciones.

Para todo este ejercicio hemos supuesto que contamos con la precondición de que `k` es distinto de `0`, para que así se pueda realizar la división. Si no fuese así,

se podría encerrar el cuerpo de ambos métodos dentro de un `if` que comprobase que `k` es, en efecto, distinto de `0`. De esta manera no se produciría una excepción.

3. Una cola con prioridad es un Tipo Abstracto de Datos similar a una cola, pero en el que los elementos tienen, además, una prioridad asignada. En una cola de este tipo, un elemento con mayor prioridad es siempre desencolado antes que otro de menor prioridad y cuando dos elementos tienen la misma prioridad, se desencolarán según su orden de entrada en la cola.

   Consideremos el caso en el que sabemos que existe un número finito y conocido k de prioridades posibles para los elementos (por ejemplo, para acceder a un avión suele haber sólo dos prioridades: embarque normal o prioritario) y las siguientes tres representaciones posibles:

   CP1) Utilizar una lista no ordenada para almacenar los elementos en el orden en el que van llegando a la cola.

   CP2) Utilizar una lista similar a la anterior, pero ordenada según la prioridad de los elementos (y el orden en el que se añaden en caso de igualdad de prioridades), de forma que un elemento que deba ser desencolado antes que otro deberá estar situado en un índice inferior en la lista.

   CP3) Agrupar los elementos en k colas, de forma que cada una de ellas almacene elementos con igual prioridad.

   a) (1.5 Puntos) Razone justificadamente el coste asintótico temporal en el caso peor de la operación de añadir un nuevo elemento a la cola de prioridad en cada una de las tres representaciones. Preste especial atención a los factores que pueden intervenir en el tamaño del problema.

      CP1) Dado que la estructura donde se almacenan los elementos (junto a su prioridad) es una lista sin ningún tipo de orden, nada nos impide insertar los nuevos elementos en la primera posición de la lista. Esa operación tendría un coste constante con respecto al número de elementos de la lista.

      Esta estrategia nos sitúa los elementos que acaban de llegar a la cola en la primera posición y los primeros elementos en entrar en la cola en la última posición, es decir, ordenamos al revés los elementos. Otra solución podría ser insertar el elemento en la última posición de la lista, lo cual tendría un coste lineal con respecto al número de elementos contenidos en la lista.

      Por lo cual, sin saber cuál de las dos formas de inserción se utiliza, en el peor de los casos el coste sería lineal con respecto al número de elementos almacenados.

      Como veremos más adelante, la eliminación del elemento más prioritario va a implicar un recorrido completo de la lista independientemente del orden en el

que almacenemos los elementos, por lo que las dos opciones propuestas aquí serán perfectamente válidas.

CP2) Para realizar la inserción será necesario ir recorriendo la lista y comparar la prioridad del nuevo elemento con la del siguiente elemento de la lista. El punto de inserción será aquel en el que el siguiente elemento de la lista tenga una prioridad menor.

En el peor de los casos, estaríamos insertando un elemento de la mínima prioridad, lo que haría que tuviéramos que recorrer toda la lista, por lo que el coste asintótico temporal en el caso peor es lineal con respecto al número de elementos contenidos en la lista.

CP3) En este caso, deberemos insertar el elemento como el último de la cola que almacena los elementos con igual prioridad. El coste será constante con respecto al número total de elementos almacenados, pero dependiendo de cómo se almacenen las k colas el coste podría ser constante con respecto a k (si pudiéramos acceder directamente a la cola adecuada) o lineal con respecto a k (si tuviéramos que buscar dicha cola).

En el peor de los casos, el coste sería lineal con respecto a k.

b) (1.5 Puntos) Razone justificadamente el coste asintótico temporal en el caso peor de la operación de eliminar el elemento de mayor prioridad en cada una de las tres representaciones.

CP1) Dado que los elementos se están almacenando sin ningún tipo de orden con respecto a su prioridad, en el peor de los casos, el elemento más prioritario estará situado al final de la lista (con independencia de si introducimos los elementos nuevos al comienzo o al final de la misma).

Así pues, el coste de esta operación para este caso será lineal con respecto al número total de elementos almacenados en la lista.

CP2) Tal y como está administrada esta estructura, el elemento con mayor prioridad estará siempre situado como primer elemento de la lista, por lo que eliminarlo tendrá un coste constante con respecto al número de elementos almacenados.

CP3) En este caso, eliminar el elemento de mayor prioridad consiste en eliminar el primer elemento de la cola que contenga los elementos (es decir, que no esté vacía) de mayor prioridad presentes en ese momento en la estructura.

Para ello habrá que consultar por orden de prioridad las k colas para comprobar si están o no vacías y eliminar el primer elemento de la primera que no esté vacía. Este proceso es lineal con respecto a k (en el peor de los casos la única

cola no vacía será la que almacene los elementos de menor prioridad) y constante con respecto al número total de elementos y al número de elementos que estuvieran almacenados en esa cola.

Por lo tanto, el coste es lineal con respecto a k.

c) (0.5 Puntos) En cuanto al espacio necesario en memoria, ¿es el mismo en las tres representaciones?

Para las dos primeras representaciones, se requiere almacenar para cada elemento no sólo su valor, sino también su prioridad, ya que en la misma estructura se mezclan valores de diferentes prioridades. La única diferencia entre ambas representaciones es que CP2 prescribe un orden entre elementos, mientras que CP1 no.

En cuanto a CP3, si todos los elementos con igual prioridad se almacenan en una cola común, no será necesario asociar a cada elemento con su prioridad de forma independiente, pudiendo "*sacar factor común*" y asociar toda la cola a la prioridad de todos sus elementos.

Por lo tanto, en este caso el coste en memoria sería menor que en los otros dos.

d) (0.5 Puntos) Si sabemos que el número de niveles de prioridad es mucho menor que el número de elementos esperado, ¿cuál sería la mejor representación? ¿Y en el caso contrario?

Llamemos n al número de elementos esperado. En vista de las respuestas de los apartados anteriores, podemos crear la siguiente tabla:

	Inserción	**Extracción**	**Memoria**
CP1	O(n)	O(n)	Igual CP1 y CP2
CP2	O(n)	O(1)	Igual CP1 y CP2
CP3	O(k)	O(k)	Menor que CP1 y CP2

Por lo que, si k << n (notación matemática que se lee "*k es mucho menor que n*"), la mejor representación sería CP3 y, en caso contrario (k >> n) sería CP2, ya que la representación CP1 es fácil de descartar pues emplea la misma memoria que CP2, pero la extracción es siempre más costosa.

Hay que notar en este punto que si estuviéramos en el caso en el que k >> n y se supiese que en la representación CP1 los nuevos elementos se almacenasen al comienzo de la lista, no podríamos decidir si CP1 o CP2 sería la representación más adecuada, pues el coste de una operación es constante en un caso y lineal en el otro. Llegados a ese punto, tendríamos que decidir en base a cuál sería la

operación crítica (inserción o extracción), para elegir la representación con menor coste para dicha operación.