P1. **Pregunta sobre la práctica**. Se quiere dotar al deposito de queries de un histórico, de manera que sólo se almacenen las últimas `n` queries individuales introducidas (siendo `n` un número conocido que se le pasa por parámetro al constructor).

a) (1 Punto) Describa (no es necesario programar) cómo puede representarse dicho histórico mediante los TADs estudiados en la asignatura. Justifique su respuesta.

Nos indican que se desean almacenar las últimas queries individuales, es decir, el texto de las mismas. Por lo cual podríamos almacenar únicamente ese texto (en forma de cadena de caracteres), sin necesidad de almacenar objetos de la clase Query.

Por otro lado, cuando tengamos almacenadas ya las últimas n queries en el histórico, al almacenar la siguiente deberemos eliminar la que más tiempo lleve en el histórico, es decir: la primera que se hubiera añadido. Esto concuerda con una política de gestión FIFO: el primero que entra es el primero en salir, lo que nos lleva a considerar que la estructura más adecuada para representar este histórico sería una cola de cadenas de caracteres. Además será necesario almacenar el valor máximo que puede contener dicha cola, porque tendremos que consultarlo para poder saber si hay que eliminar el elemento más antiguo o no.

Dado que la política identificada es FIFO, descartamos inmediatamente el uso de una pila, ya que su política de gestión es LIFO y sería complicado utilizarla para nuestros propósitos. Nos quedaría como posibilidad el uso de una lista, pero también lo descartamos por motivos de eficiencia, ya que si bien el acceso al primer elemento de la lista se realizaría en tiempo constante, es necesario también acceder al último elemento, lo cual nos lleva un tiempo lineal con respecto al tamaño del histórico, mientras que con una cola el acceso al primer y último elemento se realizan en tiempo constante.

Por lo tanto, para almacenar el histórico sería necesario añadir una cola de cadenas de caracteres y un valor entero a la estructura de datos que ya estuviéramos utilizando para la representación del repositorio de queries (lista o árbol).

b) (1 Punto) Identifique las operaciones de `QueryDepotIF` que se verían afectadas por la inclusión del histórico y detalle cómo deberían modificarse (no es necesario programarlas).

Las operaciones de consulta (`numQueries`, `getFreqQuery` y `listOfQueries`) no se ven afectadas por la inclusión de este histórico, pues su tarea seguirá involucrando acceder al repositorio de queries.

De entre las operaciones detalladas en el interfaz, la única que se ve afectada es la operación de modificación `incFreqQuery`. Esta operación deberá seguir realizando el mismo trabajo que realizaba antes de la inclusión del histórico: incrementar, de forma adecuada a la estructura empleada, la frecuencia de la query asociada a la cadena de caracteres recibida (e incluyéndola en el repositorio si no estaba previamente).

Pero además de eso deberá añadir la nueva cadena al histórico de consultas, metiéndola en la cola, para lo cual deberá comprobar si la cola tiene ya el tamaño máximo, en cuyo caso debería eliminar el primer elemento, para que no se almacenasen más consultas de las deseadas.

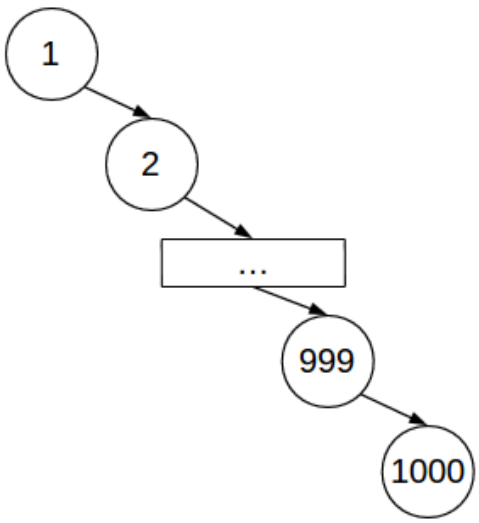
Adicionalmente, hay que hacer notar que también se ve afectado el constructor de la clase, aunque no está definido en el interfaz y no se tiene en cuenta para la corrección del examen. Dicho constructor deberá añadir un parámetro que represente el número máximo de consultas a almacenar en el histórico y deberá ocuparse de inicializar correctamente los dos nuevos atributos añadidos (cola y entero).

1. Se pide:

   a) (1 Punto) ¿Cuál es el coste asintótico temporal en el caso peor de localizar un elemento cualquiera en un **árbol de búsqueda binaria**? ¿Por qué? Justifique su respuesta.

   Un árbol de búsqueda binaria cumple la propiedad de que todos los elementos mayores que la raíz están en un hijo y que todos los elementos menores están en el otro hijo. Además ambos hijos serán, a su vez, árboles de búsqueda binaria.

   Sin pérdida de generalidad, vamos a suponer que el árbol estuviera ordenado de manera creciente, lo que situaría a todos los elementos menores que la raíz en el hijo izquierdo y los mayores en el derecho. Ahora supongamos que en dicho árbol se introducen los elementos del 1 al 1000 en orden creciente. Esto significará que el aspecto final del árbol será el siguiente:



Es decir, cada nodo padre tiene un único hijo derecho. Esto es lo que se denomina un árbol degenerado, ya que ha perdido su estructura jerárquica y se asemeja a una estructura lineal de lista.

Si ahora quisiéramos acceder a cualquiera de los elementos, tendríamos que recorrer todos los anteriores. Así, para acceder al elemento 1000, tendríamos que recorrer los 999 primeros. Este ejemplo puede generalizarse a un número arbitrario n de elementos, lo que nos indica que el coste de localizar un elemento cualquiera será lineal con respecto al número de elementos contenidos en el árbol.

b) (1 Punto) ¿Cómo consiguen los **árboles AVL** mejorar este coste? Justifique su respuesta.

Los árboles AVL son una especialización de los árboles de búsqueda binaria que, además de cumplir la propiedad de éstos (ver apartado anterior) cumplen que la diferencia entre alturas de cada uno de sus hijos es, a lo sumo, 1. Claramente el árbol anterior no lo cumple, por lo que no sería un árbol AVL a pesar de sí ser un árbol de búsqueda binaria.

En un árbol AVL es de esperar que el número de nodos contenidos en cada uno de sus hijos sea, aproximadamente, la mitad del número de nodos totales contenidos en el árbol. Esto hace que al aplicar la búsqueda binaria, podamos reducir el problema mediante división, con un factor de reducción de 2 (cada llamada recursiva supondrá buscar el elemento en un árbol con la mitad de nodos que la anterior llamada). Esto significa que el coste de localizar un elemento determinado será logarítmico con respecto al número total de nodos contenidos en el árbol, lo cual es una mejora sustancial con respecto al orden lineal que podemos garantizar en un árbol de búsqueda binaria genérico, tal y como vimos en el apartado anterior.

2. (2 Puntos) Se tienen dos pilas de enteros que están ordenadas de menor a mayor (es decir, el número más pequeño está en la cima de cada pila). Se desea programar un método:

```
public StackIF<Integer> mixStacks( StackIF<Integer> s1,
                                   StackIF<Integer> s2 );
```

que combine las pilas `s1` y `s2` en una nueva pila de enteros que contenga todos los elementos presentes en `s1` y `s2` y que esté ordenada de menor a mayor. No puede utilizar iteradores y las pilas pasadas por parámetro deben conservar su contenido a la salida del método.

En primer lugar vamos a detallar el funcionamiento del método, para luego pasar a ver el código completo del mismo.

Como las pilas siguen una política de gestión LIFO, para conservar el orden de los elementos deberemos invertirlas, así que ese será el primer paso: invertir ambas pilas. Para ello creamos dos pilas auxiliares (`sAux1` y `sAux2`) y vamos metiendo en ellas los elementos que sacamos de las originales (`s1` y `s2` respectivamente).

Llegados a este punto, tenemos las pilas originales vacías y las pilas auxiliares con el contenido de las originales invertido. Es decir, las pilas auxiliares estarán ordenadas de mayor a menor.

Ahora creamos una nueva pila (`mixed`) donde vamos a mezclar ordenadamente los elementos de las pilas auxiliares `sAux1` y `sAux2`. Para ello realizaremos un bucle que se ejecutará mientras ambas pilas no estén vacías.

¿Qué haremos en el interior de este bucle? Primero comprobamos si la pila `sAux1` es vacía, en cuyo caso sabemos que `sAux2` no lo será. Por tanto, apilaremos el valor contenido en la cima de `sAux2` tanto en `mixed` (para devolverlo) como en `s2` (para restaurar el contenido de dicha pila). Una vez hecho esto, eliminaremos ese elemento de `sAux2`.

Si `sAux1` resulta ser no vacía, comprobaremos si `sAux2` lo es, en cuyo caso haremos lo mismo que antes, pero con el elemento en la cima de `sAux1`, que lo apilaremos tanto en `mixed` como en `s1`.

Si tanto `sAux1` como `sAux2` fuesen no vacías, entonces deberemos comparar los elementos situados en la cima de ambas (`t1` y `t2` respectivamente). Si `t1 > t2`, significa que `t1` ha de ser apilado en `mixed` antes que `t2` (para que al final esté ordenada de menor a mayor), por lo cual, lo apilamos en `mixed` (y en `s1` para restaurar el valor de dicha pila). En caso contrario, apilaremos `t2` tanto en mixed como en `s2`.

En cualquiera de los dos casos, sacamos el elemento apilado en `mixed` de la pila auxiliar correspondiente y cerramos el bucle. De esta forma, el resultado correcto de la mezcla se ha creado en la pila `mixed` y tanto `s1` como `s2` no ven modificado su contenido.

```
public StackIF<Integer> mixStacks( StackIF<Integer> s1,
                                   StackIF<Integer> s2 ) {
     StackIF<Integer> sAux1 = new Stack<Integer>();
     while ( !s1.isEmpty() ) {
          sAux1.push(s1.getTop());
          s1.pop();
     }
     StackIF<Integer> sAux2 = new Stack<Integer>();
     while ( !s2.isEmpty() ) {
          sAux2.push(s2.getTop());
          s2.pop();
     }
```

```
        StackIF<Integer> mixed = new Stack<Integer>();
        while ( !(sAux1.isEmpty() && sAux2.isEmpty()) ) {
             if ( sAux1.isEmpty() ) {
                  mixed.push(sAux2.getTop());
                  s2.push(sAux2.getTop());
                  sAux2.pop();
             } else {
                  if ( sAux2.isEmpty() ) {
                       mixed.push(sAux1.getTop());
                       s1.push(sAux1.getTop());
                       sAux1.pop();
                  } else {
                       Integer t1 = sAux1.getTop();
                       Integer t2 = sAux2.getTop();
                       if ( t1 > t2 ) {
                            mixed.push(t1);
                            s1.push(t1);
                            sAux1.pop();
                       } else {
                            mixed.push(t2);
                            s2.push(t2);
                            sAux2.pop();
                       }
                  }
             }
        }
        return mixed;
   }
```

Otra forma de hacerlo es recursivamente, en cuyo caso no sería necesario invertir las pilas, ya que se vaciarían en las llamadas recursivas y se reconstruirían a la vuelta de las mismas. Dejamos esa variante como ejercicio.

3. Un **Multiconjunto** es una colección que permite almacenar elementos y en la que puede haber más de una instancia de cada uno de ellos. Al número de repeticiones de un elemento se le llama multiplicidad.

   Nota: recuerde que dispone del interfaz de Multiconjunto en las hojas de interfaces de este enunciado.

   Consideremos los siguientes escenarios:

   > MC1) Sabemos que los elementos pueden ordenarse, por lo que utilizamos una lista ordenada (crecientemente mediante el orden de los elementos) de pares `<elemento,multiplicidad>` como estructura de datos para implementar un Multiconjunto.
   >
   > MC2) Sabemos que los elementos pueden ordenarse, por lo que utilizamos un árbol AVL (ordenado crecientemente mediante el orden de los elementos) de pares `<elemento,multiplicidad>` como estructura de datos para implementar un Multiconjunto.

MC3) No sabemos si los elementos pueden ordenarse o no, por lo que utilizamos una lista no ordenada de pares `<elemento,multiplicidad>` para implementar el Multiconjunto.

a) (1.5 Puntos) Razone justificadamente el coste asintótico temporal en el caso peor del método `addMultiple(e,n)` (ver interfaz) en cada una de las tres representaciones. Preste especial atención a los factores que pueden intervenir en el tamaño del problema.

Con independencia de la representación escogida, el método addMultiple(e,n) ha de buscar en el Multiconjunto el elemento e. Si existe, modificará su multiplicidad sumándole n, mientras que si no existe, añadirá ese elemento al Multiconjunto con multiplicidad n.

Así pues, el coste del método va a ser el coste de la búsqueda más la modificación de la multiplicidad o de la búsqueda más la inserción del nuevo elemento. Sin pérdida de generalidad, supondremos que la comparación entre elementos se puede realizar en tiempo constante. Si no fuera así, todos los costes se verían afectados por el coste de la comparación (que multiplicaría al coste de las búsquedas).

MC1) La búsqueda de un elemento en una lista tiene, en el peor de los casos, un coste asintótico lineal con respecto al número de elementos contenidos en la lista, ya que en el peor de los casos buscaríamos el último elemento.

Una vez localizado el elemento, la modificación se puede realizar en tiempo constante, por lo que si el elemento ya existía en el Multiconjunto, el coste será lineal con respecto al número de elementos diferentes contenidos en el Multiconjunto.

Si el elemento no estuviera, al estar la lista ordenada podemos cortar la búsqueda cuando nos encontramos un elemento mayor que el buscado. En ese caso la búsqueda también será lineal, al igual que la inserción en el punto adecuado.

Por lo tanto, con la estructura de lista ordenada, el coste de `addMultiple(e,n)` será lineal con respecto al número de elementos diferentes contenidos en el Multiconjunto.

MC2) Al utilizar un árbol de búsqueda binaria, la localización de un elemento tiene un coste logarítmico con respecto al número de elementos contenidos en el árbol.

Si el elemento existe, la modificación se realiza en tiempo constante (pues hemos localizado el elemento y se modifica la frecuencia, lo que no influye en la ordenación del árbol) y, si no existe, localizar el punto donde ha de insertarse el elemento también tiene un coste logarítmico y la inserción puede involucrar a lo sumo una rotación doble, lo cual se realiza en tiempo constante.

Por lo tanto, utilizando un AVL como estructura de soporte, el método `addMultiple(e,n)` tendría un coste logarítmico con respecto al número de diferentes elementos que se encuentran en el Multiconjunto.

MC3) Se puede utilizar casi la misma argumentación que utilizamos en MC1 para concluir que el coste asintótico de addMultiple(e,n) en este caso sería también lineal con respecto al número de elementos distintos que contiene el Multiconjunto.

La única diferencia está en el hecho de que si un elemento no está, entonces habremos recorrido toda la lista sin encontrarlo. La inserción podría realizarse en la primera posición de la lista (operación de coste constante), ya que el orden no importa.

b) (2 Puntos) Explique cómo se podría implementar el método `union(s)` (ver interfaz) en cada una de las tres representaciones, teniendo en cuenta las características de cada una de ellas. Justifique su respuesta.

MC1) La idea consiste en recorrer simultáneamente las listas ordenadas de pares `<elemento, multiplicidad>` tanto del Multiconjunto llamante como del Multiconjunto parámetro comparando los siguientes elementos a recorrer:

- Si el elemento más pequeño está en la lista del llamante, significa que dicho elemento sólo está en el Multiconjunto llamante, por lo que su multiplicidad no se vería afectada al realizar la unión.
- Si el elemento más pequeño está en la lista del parámetro, significa que dicho elemento sólo está en el Multiconjunto parámetro, por lo que habría que añadirlo (en la posición correspondiente) a la lista del Multiconjunto llamante.
- Si ambos elementos son iguales, significa que están en ambos Multiconjuntos, por lo que incrementamos la frecuencia del elemento en el Multiconjunto llamante con la frecuencia del elemento en el Multiconjunto parámetro.

En cualquiera de los casos, avanzamos el recorrido de la lista que tuviera el elemento más pequeño (ambas en el caso de que sean iguales).

MC2) Podemos emplear un algoritmo similar al descrito en el caso anterior sin más que "secuencializar" el contenido de los árboles mediante un iterador en inorden. La única diferencia estaría en el hecho de que en lugar de insertar en una lista, deberíamos insertar en un AVL. De esta forma aprovechamos el hecho de que los elementos están almacenados en orden.

Este algoritmo supondría un coste lineal con respecto a la suma de elementos de ambos AVL para crear los iteradores y luego el recorrido de ambos insertando/modificando los elementos. Esto supondría un coste `O(m+n)`, siendo

`m` el número de elementos en el AVL llamante y `n` el número de elementos en el AVL parámetro. En el peor de los casos habría que insertar todos los elementos del AVL parámetro en el llamante, lo que supondría un coste $O(n \cdot \log_2 m)$, ya que la inserción en un AVL es logarítmica con respecto al número de elementos que contiene. Como este coste es superior al de la creación de los iteradores, sería el del algoritmo.

Otra forma de hacerlo sería olvidarnos del iterador del AVL llamante y generar sólo el del AVL parámetro (coste $O(n)$) y para cada elemento en dicho AVL buscarlo en el AVL llamante para modificarlo o insertarlo si no estaba. Como esas operaciones tienen un coste en $O(\log_2 m)$, tendríamos que el coste sería $O(n \cdot \log_2 m)$, con lo que este algoritmo es igual de costoso que el anterior, desde un punto de vista del coste asintótico. Sin embargo, nos ahorraríamos la creación de un iterador, por lo que su coste real sería menor.

MC3) Dado que en este caso no hay un orden para los elementos, la única forma de proceder es recorrer la lista del Multiconjunto parámetro y para cada par `<e,n>` perteneciente a ella, ejecutar un `addMultiple(e,n)` en el Multiconjunto llamante.

Este último algoritmo también podría ser utilizado en los escenarios MC1 y MC2, aunque es menos eficiente que los sugeridos anteriormente al no tener en cuenta el hecho de que los elementos se almacenan ordenados en dichos escenarios.

c) (0.5 Puntos) En base a las respuestas dadas en los apartados anteriores, ¿cuál considera que sería la mejor representación si los elementos pueden ordenarse? Justifique su respuesta

En el caso del método `addMultiple`, ya hemos visto que el coste asintótico temporal en el caso peor es mejor en el escenario MC2 (donde utilizamos un AVL).

En cuanto al coste asintótico temporal en el caso peor del método `union` también será mejor en el caso del AVL, ya que (utilizando el primer algoritmo descrito para el escenario MC2) en ambos casos será necesario recorrer todos los elementos de ambos Multiconjuntos, pero la inserción de nuevos elementos en un AVL se realiza en tiempo logarítmico con respecto al total de elementos, mientras que en una lista ordenada se emplea un tiempo lineal con respecto al número total de elementos.

Por lo tanto, la representación más adecuada en caso de que los elementos puedan ser ordenados sería un árbol AVL.

MultiSetIF (Multiconjunto)

```
public interface MultiSetIF<E> extends ContainerIF<E> {
    /* Añade varias instancias de un elemento al multiconjunto  *
     * @pre:  n > 0 && premult = multiplicity(e)                 *
     * @post: multiplicity(e) = premult + n                      */
    public void addMultiple (E e, int n);
    /* Elimina varias instancias de un elemento del             *
     * multiconjunto                                            *
     * @pre: 0<n<= multiplicity(e) && premult = multiplicity(e) *
     * @post: multiplicity(e) = premult - n                      */
    public void removeMultiple (E e, int n);
    /* Devuelve la multiplicidad de un elemento dentro del      *
     * multiconjunto.                                           *
     * @return: multiplicidad de e (0 si no está contenido)     */
    public int multiplicity (E e);
    /* Realiza la unión del multiconjunto llamante con el       *
     * parámetro                                                */
    public void union (MultiSetIF<E> s);
    /* Realiza la intersección del multiconjunto llamante con   *
     * el parámetro                                             */
    public void intersection (MultiSetIF<E> s);
    /* Realiza la diferencia del multiconjunto llamante con el  *
     * parámetro (los elementos que están en el llamante pero   *
     * no en el parámetro                                       */
    public void difference (MultiSetIF<E> s);
    /* Devuelve cierto sii el parámetro es un submulticonjunto  *
     * del llamante                                             */
    public boolean isSubMultiSet (MultiSetIF<E> s);
}
```