Estrategias de Programación y Estructuras de Datos

Dpto. de Lenguajes y Sistemas Informáticos

Material permitido: NINGUNO. Duración: 2 horas

Alumno:	**D.N.I.:**
C. Asociado en que realizó la Práctica Obligatoria:	

Este documento detalla una posible solución propuesta por el Equipo Docente con intención didáctica antes que normativa. Nótese que el nivel de detalle de este documento no es el pretendido para los ejercicios realizados por los alumnos en la solución de sus exámenes.
Por brevedad, no se incluyen las interfaces de los tipos, que el alumno puede consultar en la documentación de la asignatura.

P1 Práctica. Se quiere mantener un depósito donde se almacenen las k queries más frecuentes de forma que el acceso a la i-ésima query más frecuente mediante el metodo:

```
public Query getMostFrequentQuery (int i);
```

tenga un coste asintótico temporal en el caso peor de orden lineal con respecto a k.

Recordamos que las operaciones de la interfaz `QueryDepot` son:

QueryDepot:

```
public interface QueryDepot {
  //Constructor vacío
  public QueryDepot ();
  //Constructor que recibe una lista de queries
  public QueryDepot (ListIF<String> queries);
  public int numQueries ();
  public int getFreqQuery (String q);
  public ListIF<Query> listOfQueries (String prefix);
  public void incFreqQuery (String q);
  public void decFreqQuery (String q);
}
```

[**Nota**: Se ha cambiado la situación del recordatorio anterior en este documento respecto al enunciado original para mantener la coherencia en la lectura del mismo en el marco de esta solución.]

a) (0'5 puntos) Describa brevemente qué estructura de datos es la más adecuada para poder almacenar dicho depósito de queries y qué características debe cumplir.

Para almacenar las k queries más frecuentes (nótese que a igualdad de frecuencia en las *últimas* posiciones de entre esas k, no importa cuáles se elijan y cuáles no) basta con añadir un nuevo atributo: una lista estática de longitud k que deberá preservar el orden de las queries más frecuentes según su valor de frecuencia:

```
public ListStatic<Query> mostFrequentQueries;
```

De este modo, el método `getMostFrequentQuery()` indicado en el examen podría implementarse con coste lineal respecto a k tal y como se especifica, puesto que consistirá en recorrer la lista y comparar cada query con la pasada por parámetro. Nótese que el uso de otras esructuras lineales de almacenamiento (pilas o colas) estaría desaconsejado, ya que éstas funcionan manteniendo su orden (implícito) de llegada de los elementos a las mismas según sus respectivas políticas de acceso.

b) (1'5 puntos) Enumere las operaciones de `QueryDepot` que se verían afectadas al incorporar el anterior depósito de queries y describa detalladamente cuál sería el efecto y cómo deben modificarse dichas operaciones (no es necesario implementar código).

A continuación se enumeran las operaciones afectadas por la introducción del depósito de las queries más frecuentes:

- Constructor: Debe tener como nuevo parámetro un entero k que servirá para inicializar la lista estática con la que se representa el depósito de las queries más frecuentes.
- `incrFrecQuery(Query q)`: Este método introduce una nueva query q con frecuencia 1 en caso de que la query no estuviese en el depósito general, o incrementa en 1 la frecuencia de la query q que previamente está en el depósito general. Tras realizar esta operación, debe comprobar que la frecuencia de q supera la menor frecuencia de las queries del nuevo depósito, manteniendo ordenadas las queries más frecuentes: si q estaba en el depósito, ello exigiría desplazarla a su posición adecuada. Si no, se requeriría inserción ordenada (ojo, no reordenación) y extracción de la última query (la de menor frecuencia) que hubiese en la lista de más frecuentes, en caso de que esta lista ya estuviese completa con k queries. Esto consigue mantener la propiedad de que la operación `getMostFrequentQuery` tenga coste lineal respecto a k como se pide en el enunciado.
- `decFreqQuery(Query q)`: Este método decrementa en 1 la frecuencia de la query q pasada por parámetro. Nótese que esta operación podría anular dicha frecuencia, en cuyo caso, habría que eliminar la query de la lista de más frecuentes (lo que podría motivar la necesidad de entrada de una nueva query en dicha lista de más frecuentes) así como del propio depósito o almacén (como se especificaba en el enunciado). De no resultar nula la nueva frecuencia, se debe comprobar si q estaba en el depósito de más frecuentes. En caso negativo, no es necesario hacer ninguna operación adicional. En caso afirmativo, hay que recolocar q según su frecuencia (nótese que esta operación no implica reordenación completa de la lista) y comprobar si q queda como la última de las k más frecuentes y con frecuencia estrictamente menor que la penúltima. En dicho caso, se ha de buscar una query que no esté en el depósito de más frecuentes pero que tenga mayor frecuencia que q. Si dicha query existe, entonces ésta substituirá a q.
- Los métodos de operaciones (frecuencias y sugerencia) no se ven afectados por la introducción del depósito de queries más frecuentes.

1. Se desea un método tal que, tomando como precondición que todos los números de la lista sean no negativos, devuelva el mayor prefijo cuya suma no supere n. Por ejemplo, la llamada:

```
maxPrefix([4,5,0,3,2,1],10);
```

devolvería $[4, 5, 0]$ puesto que dicho prefijo suma $9 \leq 10$ y el siguiente prefijo $[4, 5, 0, 3]$ suma $12 > 10$.

a) (1'5 puntos) Implementar de **manera recursiva** en JAVA una operación que responda a la descripción anterior y cuya signatura sea:

```
ListDynamic<Integer> maxPrefix(ListDynamic<Integer> l, int n)
```

Una forma de resolver el ejercicio consiste en aplicar la técnica de inmersión: se define un método inmersor más abstracto que el original (que se dice sumergido) con un parámetro adicional al pedido en el enunciado que irá acumulando la suma (por tanto, se trata de una inmersión de parámetros por acumulación). El método pedido en el enunciado será el método sumergido que consistirá en llamar al método inmersor inicializando sus parámetros convenientemente, es decir, concretando para qué valor de los nuevos parámetros inmersores el método que estamos construyendo devuelve el mismo valor que se espera del método sumergido. Véase que, dado que el valor original del parámetro entero no se necesita más que para la invocación inicial, otro posible desarrollo consistiría en utillizarlo restando del mismo el valor de cada elemento añadido al prefijo en cada nueva llamada interna.

El método inmersor es el siguiente:

```
/* @param l la sublista aún por procesar                                  */
/* @param n el valor máximo que no debe superarse                         */
/* @param s parámetro de inmersión que acumula el valor sumado en el      */
/*         prefijo ya procesado (es decir, la lista original menos el     */
/*         sufijo l)                                                      */
public ListDynamic<Integer> maxPrefix(ListDynamic<Integer> l,
                                      int n, int s){
  ListIF<Integer> empty = new ListDynamic<Integer>;
  if (l.isEmpty()){
    return empty;
  }
  if (s+l.getFirst()>n){
   return empty;
  }
  return maxPrefix(l.getTail(),n,s+l.getFirst()).insert(l.getFirst());;
}
```

El método sumergido es el pedido en el enunciado inicializando el parámetro adicional que va calculando la suma del prefijo que cumple la propiedad a 0, que es el valor al que se inicializa, dado que aún no se ha realizado ninguna suma respecto al entero introducido:

```
  public ListDynamic<Integer> maxPrefix(ListDynamic<Integer> l, int n){
    return maxPrefix(l,n,0);
  }
```

b) (0'5 puntos) ¿Cuál es el coste asintótico temporal de su solución en el caso peor? Describa brevemente la razón.

Sea m el tamaño de la lista l pasada por parámetro. El caso peor se da cuando el máximo prefijo coincide con la propia lista l (es decir, la suma de la lista no supera el valor n dado como parámetro) porque es cuando el método anterior realiza el máximo número de llamadas recursivas, esto es, m llamadas recursivas (dado que sólo se realiza una invocación interna por cada llamada externa al método). En cada una de las llamadas recursivas se llevan a cabo operaciones que no dependen del tamaño de la lista, por lo que pueden considerarse de coste constante respecto al mismo, por lo que el coste del método estará en $\mathcal{O}(m)$.

2. En las guías de estudio de cada asignatura de un grado existe un apartado de prerrequisitos para poder cursar la asignatura convenientemente. Dichos prerrequisitos consisten en la recomendación de haber cursado determinadas asignaturas del cuatrimestre anterior previamente. Por ejemplo, los prerrequisitos de la asignatura de primer cuatrimestre de segundo curso en ambos grados de Informática en la UNED "Programación y Estructuras de Datos Avanzadas" son haber cursado con anterioridad "Programación Orientada a Objetos" y "Estrategias de Programación y Estructuras de Datos (EPED)", ambas de segundo cuatrimestre de primer curso.

   Estos prerrequisitos definen una jerarquía entre las asignaturas del Grado. Así, los prerrequisitos inmediatos de una asignatura (como los vistos en el párrafo anterior) se considerarán *directos*. Los prerrequisitos de los prerrequisitos de una asignatura, serán *indirectos*. Por ejemplo, si un prerrequisito de "Teoría de los Lenguajes de Programación (TLP)" (segundo cuatrimestre de segundo curso) es "Programación y Estructuras de Datos Avanzadas", entonces, un prerrequisito indirecto de TLP será EPED, que, a su vez, será prerrequisito indirecto de cualquier asignatura que tenga a TLP como prerrequisito.

La siguiente interfaz, denominada `DegreePrerequisitesIF` modela la jerarquía de prerrequisitos de un Grado:

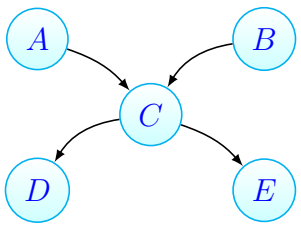
`DegreePrerequisitesIF`:

```
//Representa la jerarquía de prerrequisitos de un grado
public interface DegreePrerequisitesIF{

/* Añade el prerrequesito de que para cursar la asignatura s1 debe  */
/* haberse cursado previamente la asignatura s2. [0.5 puntos]       */
public void addPrerequisite(String subject1,String subject2);

/* Elimina la asignatura dada por parámetros de la jerarquía de     */
/* prerrequisitos. Pista: La eliminación de una asignatura puede    */
/* conllevar cambios en la lista de prerrequisitos del resto de     */
/* asignaturas [1 punto]                                            */
public void remove(String subject);

/* Devuelve una lista con TODOS los prerrequisitos de la asignatura */
/* dada por parámetro: sus prerrequisitos directos e indirectos     */
/* (prerrequisitos ancestros de la asignatura) [1 punto]            */
public ListIF<String> getPrerequisites(String subject);

/* Devuelve una lista con las asignaturas que piden como            */
/* prerrequisito haber cursado previamente la asignatura dada por   */
/* parámetro [1.5 puntos]                                           */
public ListIF<String> getSubjects(String subject);
}
```

Se pide:

a) (1 punto) Describa detalladamente cómo realizaría la representación interna de este tipo de datos **usando los TADs vistos en la asignatura**. Justifique su respuesta. Implemente un constructor de una clase que implemente la interfaz `DegreePrerequisitesIF` de manera que reciba una lista de objetos de tipo `String` con los nombres de las asignaturas.

La clase `DegreePrerequisites` que implemente la interfaz `DegreePrerequisitesIF` pedida en el enunciado puede verse como una lista de asignaturas asociadas con las listas de sus correspondientes requisitos directos. Para recoger esta información, es conveniente disponer de una clase auxiliar, que llamaremos `Subject`, que asocia una asignatura mediante su nombre (una cadena de caracteres) con su lista de prerrequisitos (que serán otras asignaturas). Así pues,la clase `DegreePrerequisites` se podrá representar como una lista de objetos de la clase (`Subject`), de manera que cada uno de estos objetos se corresponda con una asignatura diferente. Nótese que tanto en la lista de prerrequisitos de una asignatura como en la de asignaturas el orden no es relevante. Véase también que, a pesar de que el enunciado refiere una jerarquía definida entre las asignaturas por la relación de *ser prerrequisito de*, no es conveniente el uso de estructuras arborescentes para su representación en este problema, ya que pueden existir tanto asignaturas con más de un requisito como asignaturas que sean requisito de otras varias. Ambas condiciones a un tiempo impiden modelar esta jerarquía como un árbol. Veamos un ejemplo gráfico donde la asignatura C cumple las dos condiciones anteriores:



Veamos la representación de la clase auxiliar:

```
public class Subject {
  private String name;
  private ListIF<Subject> requisites;
  ...
}
```

Y en función de ésta, la clase que necesitamos implementar:

```
public class DegreePrerequisites {
  private ListIF<Subject> subjects;
  ...
}
```

Un posible constructor de `Subject` puede recibir únicamente el nombre de la asignatura, en cuyo caso, debería inicializar su lista de prerrequisitos a la lista vacía:

```
public Subject(String subject){
  name = subject;
  requisites = new ListIF<Subject>();
}
```

Nótese que podría incluirse otro constructor que recibiese un nombre de asignatura y su lista de requisitos directos por razones de comodidad del cliente del TAD.

Los métodos accesores y modificadores (getters y setters) se dan por supuestos.

Puesto que será necesario eliminar y añadir prerrequisitos a una asignatura dada (métodos `remove` y `addPrerequisite` de la interfaz), será conveniente que existan métodos para realizar dichas operaciones en esta clase:

```
public void addNewRequisite(Subject requisite){
  if (!requisites.contains(requisite)){
    requisites.insert(requisite);
  }
}
public void removeRequisite (Subject requisite){
  ListIF<Subject> newRequisites = new ListIF<Subject>();
  Iterator<Subject> iter = requisites.getIterator();
  while(iter.hasNext()){
    Subject sAux = iter.getNext();
    if (!sAux.getName().equals(requisite.getName())){
      newRequisites.insert(sAux);
    }
  }
  requisites = newRequisites;
}
```

Por último, el constructor pedido de la clase `DegreePrerequisites` recibe una lista de nombres de asignatura, construye objetos de la clase `Subject` y los añade a la lista:

```
DegreePrerequisites(ListIF<String> lsbj){
  subjects = new ListIF<Subject>();
  Iterator<String> iter = lsbj.getIterator();
  while(iter.hasNext()){
    String sbj = iter.getNext();
    Subject sub = new Subject(sbj);
    subjects.insert(sub);
  }
}
```

b) (4 puntos) Basándose en la respuesta anterior, implemente en JAVA todos los métodos de una clase que implemente la interfaz `DegreePrerequisitesIF`. Nótese que en la descripción de la interfaz viene anotada la puntuación de cada uno de ellos.

Es conveniente añadir un método auxiliar (en la clase `DegreePrerequisites`) que nos sirva para decidir si existe una determinada asignatura. Dicho método lo definiremos como privado:

```
//Método auxiliar privado para decidir si existe una asignatura dada
private Subject findSubject(String subject){
  Iterator<Subject> iter = subjects.getIterator();
  while(iter.hasNext()){
    Subject s = iter.getNext();
    if (s.getName().equals(subject)){
      return s;
    }
  }
  return null;
}
```

Si el anterior método devuelve **`null`** significa que no existe la asignatura dada por parámetro en el listado de prerrequisitos. En caso contrario, se devuelve el objeto de la clase `Subject` asociado a dicha asignatura. Nótese que el método `findSubject` asume que no existen asignaturas repetidas.

Para añadir un prerrequisito, primero comprobaremos que existen las dos asignaturas relacionadas mediante la invocación al método (parámetros). Tomaremos la decisión de crearlas y añadirlas en caso contrario.

```
// Implementación addPrerequisite
public void addPreRequisite(String subject1, String subject2){
  Subject s1 = findSubject(subject1);
  if (s1 == null){
    s1 = new Subject(subject1);
    subjects.insert(s1);
  }
  Subject s2 = findSubject(subject2);
  if (s2 == null){
    s2 = new Subject(subject2);
    subjects.insert(s2);
  }
  s1.addNewRequisite(s2);
}
```

En caso de que exista la asignatura que se quiere eliminar, habrá que recorrer la lista eliminando tanto la asignatura como sus prerrequisitos. Para el resto de asignaturas (que no son ésa), habrá que eliminarla de sus listas de prerrequisitos (con el método `removeRequisite()` de la clase `Subject`).

```
//Implementación remove
public void remove(String subject){
  Subject s = findSubject(subject);
  if (s != null){
    /* elimina la asignatura de la lista de éstas tanto en su rol
    * de asignatura como en el de prerrequisito de otra(s). Esto
    * produce una nueva lista de asignaturas como resultado       */
    ListIF<Subject> newSubjects = ListIF<Subject>();
    Iterator<Subject> iter = subjects.getIterator();
    while(iter.hasNext()){
      Subject sAux = iter.getNext();
      if (!sAux.getName().equals(subject)){
        // esta asignatura debe permanecer en la estructura
        // pero eliminaremos 'subject' de sus prerrequisitos
        sAux.removeRequisite(s);
        newSubjects.insert(sAux);
      }
    }
    subjects = newSubjects;
  }
}
```

El cálculo de prerrequisitos ancestros sugiere que este método se plantee de manera recursiva puesto que se está modelando una jerarquía de prerrequisitos.

```
//Implementación getPrerequisites
public ListIF<String> getPrerequisites(String subject){
  ListIF<String> preRequisites = new ListIF<String>();
  Subject s = findSubject(subject);
  if (s != null){
    /* Iteraremos sobre los prerrequisitos directos (mantenidos en los
    * objetos de la clase Subject), dado que esta lista no cambia.  */
    Iterator<Subject> iter = s.getRequisites().getIterator();
    while(iter.hasNext()){
      String sName = iter.getNext().getName();
      // Introducimos cada requisito directo en la lista resultado
      preRequisites.insert(sName);
      // Obtenemos recursivamente todos los prerrequisitos de cada
      // requisito directo
      ListIF<String> preRequisitesAux = getPrerequisites(sName);
      // Iteramos sobre esta nueva lista de requisitos
      Iterator<String> iter2 = preRequisitesAux.getIterator();
      while(iter2.hasNext()){
        String nextPrerequisite = iter2.getNext();
        if (!preRequisites.contains(nextPrerequisite) ) {
          preRequisites.insert(nextPrerequisite);
        }
      }
    }
  }
  return preRequisites;
}
```

Se pide la lista de asignaturas que tienen como prerrequisito haber cursado previamente la asignatura dada por parámetro (como no se dice nada más, consideraremos sólo requisitos directos).

```
//Implementación getSubjects
public ListIF<String> getSubjects(String subject){
  ListIF<String> listSubjects = new ListIF<String>();
  Subject s = findSubject(subject);
  if (s != null){
    Iterator<Subject> iter = subjects.getIterator();
    while(iter.hasNext()){
      Subject sAux = iter.getNext();
      ListIF<Subject> requisites = sAux.getRequisites();
      if (requisites.contains(s)){
        listSubjects.insert(sAux.getName());
      }
    }
  }
  return listSubjects;
}
```

c) (1 punto) Calcule el coste asintótico temporal, en el caso peor, del método `getSubjects()` en su implementación indicando claramente sobre qué valor o valores calcula el coste. Justifique adecuadamente su respuesta.

En el caso peor, el método `getSubjects` realiza tantas iteraciones como número n de asignaturas hay en el listado de prerrequisitos. Por otro lado, dentro del bucle principal se hace una invocación al método `contains` del TAD Lista (interfaz `ListIF`) cuyo coste es lineal, y en este caso depende del número k de prerrequisitos de una asignatura actual. Por tanto, el coste del método `getSubjects` está en $\mathcal{O}(nk)$.