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:

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.

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).

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);
}
```

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)
   ```

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

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.

Consigne sus datos en todas las hojas que entregue. No se puntuarán respuestas sin justificar.

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 y sus prerrequisitos 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.

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.

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.

ListIF (Lista)

```
/* Representa una lista de elementos */
public interface ListIF<T>{
   /* Devuelve la cabeza de una lista*/
    *
    public T getFirst ();
    /* Devuelve: la lista excluyendo la
       cabeza. No modifica la estructura */
    public ListIF<T> getTail ();
   /* Inserta una elemento (modifica la
      estructura)
    * Devuelve: la lista modificada
    * @param elem El elemento que hay que
       añadir*/
    public ListIF<T> insert (T elem);
    /* Devuelve: cierto si la lista esta
       vacia */
    public boolean isEmpty ();
    /* Devuelve: cierto si la lista esta
       llena*/
    public boolean isFull();
    /* Devuelve: el numero de elementos de
       la lista*/
    public int getLength ();
    /* Devuelve: cierto si la lista
       contiene el elemento.
     * @param elem El elemento buscado */
    public boolean contains (T elem);
    /* Ordena la lista (modifica la lista)
     * @Devuelve: la lista ordenada
     * @param comparator El comparador de
        elementos*/
    public ListIF<T> sort (ComparatorIF<T>
       comparator);
    /*Devuelve: un iterador para la lista*/
    public IteratorIF<T> getIterator ();
}
```

StackIF (Pila)

```
/* Representa una pila de elementos */
public interface StackIF <T>{
    /* Devuelve:  la cima de la pila */
     public T getTop ();
    /* Incluye un elemento en la cima de
       la pila (modifica la estructura)
     * Devuelve: la pila incluyendo el
        elemento
     * @param elem Elemento que se quiere
        añadir */
     public StackIF<T> push (T elem);
    /* Elimina la cima de la pila
       (modifica la estructura)
     * Devuelve: la pila excluyendo la
        cabeza */
     public StackIF<T> pop ();
    /* Devuelve: cierto si la pila esta
       vacia */
     public boolean isEmpty ();
    /* Devuelve: cierto si la pila esta
       llena */
     public boolean isFull();
    /* Devuelve: el numero de elementos de
       la pila */
     public int getLength ();
    /* Devuelve: cierto si la pila
       contiene el elemento
     * @param elem Elemento buscado */
     public boolean contains (T elem);
    /*Devuelve: un iterador para la pila*/
    public IteratorIF<T> getIterator ();
}
```

QueueIF (Cola)

```
/* Representa una cola de elementos */
public interface QueueIF <T>{
    /* Devuelve: la cabeza de la cola */
     public T getFirst ();
    /* Incluye un elemento al final de la
       cola (modifica la estructura)
     * Devuelve: la cola incluyendo el
        elemento
     * @param elem Elemento que se quiere
        añadir */
     public QueueIF<T> add (T elem);
    /* Elimina el principio de la cola
       (modifica la estructura)
     * Devuelve: la cola excluyendo la
        cabeza    */
     public QueueIF<T> remove ();
    /* Devuelve: cierto si la cola esta
       vacia */
     public boolean isEmpty ();
    /* Devuelve: cierto si la cola esta
       llena */
     public boolean isFull();
    /* Devuelve: el numero de elementos de
       la cola */
     public int getLength ();
    /* Devuelve: cierto si la cola
       contiene el elemento
     * @param elem elemento buscado */
     public boolean contains (T elem);
    /*Devuelve: un iterador para la cola*/
     public IteratorIF<T> getIterator ();
}
```

TreeIF (Árbol general)

```
/* Representa un arbol general de
   elementos */
public interface TreeIF <T>{
    public int PREORDER  = 0;
    public int INORDER   = 1;
    public int POSTORDER = 2;
    public int BREADTH   = 3;
    /* Devuelve:  elemento raiz del arbol
       */
     public T getRoot ();
    /* Devuelve: lista de  hijos de un
       arbol.*/
     public ListIF <TreeIF <T>>
        getChildren ();
    /* Establece el elemento raiz.
```

```
     * @param elem Elemento que se quiere
        poner como raiz*/
     public void setRoot (T element);
    /* Inserta un subarbol como ultimo hijo
     * @param child el hijo a insertar*/
     public void addChild (TreeIF<T>
        child);
    /* Elimina el subarbol hijo en la
       posicion index-esima
     * @param index indice del subarbol
        comenzando en 0*/
     public void removeChild (int index);
    /* Devuelve: cierto si el arbol es un
       nodo hoja*/
     public boolean isLeaf ();
    /* Devuelve: cierto si el arbol es
       vacio*/
     public boolean isEmpty ();
    /* Devuelve: cierto si la lista
       contiene el elemento
     * @param elem Elemento buscado*/
     public boolean contains (T element);
    /* Devuelve: un iterador para la lista
     * @param traversalType el tipo de
        recorrido, que
     * sera PREORDER, POSTORDER o BREADTH
        */
     public IteratorIF<T> getIterator (int
        traversalType);
}
```

BTreeIF (Árbol Binario)

```
/* Representa un arbol binario de
   elementos */
public interface BTreeIF <T>{
  public int PREORDER  = 0;
  public int INORDER   = 1;
  public int POSTORDER = 2;
  public int LRBREADTH = 3;
  public int RLBREADTH = 4;
 /* Devuelve: el elemento raiz del arbol */
  public T getRoot ();
 /* Devuelve: el subarbol izquierdo o null
    si no existe */
  public BTreeIF <T> getLeftChild ();
 /* Devuelve: el subarbol derecho o null
    si no existe */
  public BTreeIF <T> getRightChild ();
 /* Establece el elemento raiz
  * @param elem Elemento para poner en la
     raiz */
  public void setRoot (T elem);
 /* Establece el subarbol izquierdo
  * @param tree el arbol para poner como
     hijo izquierdo */
  public void setLeftChild (BTreeIF <T>
     tree);
 /* Establece el subarbol derecho
  * @param tree el arbol para poner como
     hijo derecho */
  public void setRightChild (BTreeIF <T>
     tree);
 /* Borra el subarbol izquierdo */
  public void removeLeftChild ();
 /* Borra el subarbol derecho */
  public void removeRightChild ();
 /* Devuelve: cierto si el arbol es un
    nodo hoja*/
  public boolean isLeaf ();
 /* Devuelve: cierto si el arbol es vacio
    */
  public boolean isEmpty ();
 /* Devuelve: cierto si el arbol contiene
    el elemento
  * @param elem Elemento buscado */
  public boolean contains (T elem);
 /* Devuelve un iterador para la lista.
  * @param traversalType el tipo de
     recorrido que sera
    PREORDER, POSTORDER, INORDER,
       LRBREADTH o RLBREADTH */
 public IteratorIF<T> getIterator (int
    traversalType);
}
```

ComparatorIF

```
/* Representa un comparador entre
   elementos */
public interface ComparatorIF<T>{
  public static int LESS  = -1;
  public static int EQUAL = 0;
  public static int GREATER = 1;
 /* Devuelve: el orden de los elementos
  * Compara dos elementos para indicar si
     el primero es
  * menor, igual o mayor que el segundo
     elemento
  * @param e1 el primer elemento
  * @param e2 el segundo elemento   */
  public int compare (T e1, T e2);
 /* Devuelve: cierto si un elemento es
    menor que otro
  * @param e1 el primer  elemento
  * @param e2 el segundo elemento */
  public boolean isLess (T e1, T e2);
 /* Devuelve: cierto si un elemento es
    igual que otro
  * @param e1 el primer  elemento
  * @param e2 el segundo elemento */
  public boolean isEqual (T e1, T e2);
 /* Devuelve: cierto si un elemento es
    mayor que otro
  * @param e1 el primer  elemento
  * @param e2 el segundo elemento*/
  public boolean isGreater (T e1, T e2);
}
```

IteratorIF

```
/* Representa un iterador sobre una
   abstraccion de datos */
public interface IteratorIF<T>{
    /* Devuelve: el siguiente elemento de
       la iteracion */
     public T getNext ();
```

```
    /* Devuelve: cierto si existen mas
       elementos en el iterador */
     public boolean hasNext ();
    /* Restablece el iterador para volver
       a recorrer la estructura */
     public void reset ();
}
```