Estrategias de Programación y Estructuras de Datos
Septiembre 2013 – Original

Justifique todas las respuestas a sus ejercicios. No se valorarán respuestas sin justificar.

P1. Práctica. Supongamos que la pastelería tiene un suministro finito para cada tipo de tarta durante una jornada laboral. Esto implica que, en dicha jornada laboral, sólo pueden venderse t_1, t_2, ..., t_n tartas para los tipos de tarta 1, 2, ..., n, respectivamente, siendo todos los t_i mayores o iguales a cero. En caso de que a la máquina le tocase fabricar una tarta de tipo i y no quedase suministro para dicho tipo de tarta, entonces la máquina intentaría fabricar una tarta del siguiente tipo disponible por orden (ver ejemplo más abajo). Teniendo en cuenta lo anterior, se pide:

a) (0,5 puntos). Crear una nueva clase SuministroDiario que guarde la información citada anteriormente. Basta con especificar sus atributos e implementar un constructor adecuado para dicha clase. Usar para ello alguna estructura de datos vista durante el curso (no se permite el uso de arrays).

b) (1,5 puntos). Modificar adecuadamente la función *fabricarTarta*, de manera que cuando la máquina tuviese que fabricar una tarta de tipo i y t_i=0, fabrique una tarta del siguiente tipo disponible por orden, y que cuando no quede suministro para fabricar tartas de ningún tipo devuelva *false*. Deben seguir cumpliéndose las otras condiciones para poder fabricar una tarta: la máquina no está llena y el cliente tiene una paciencia estrictamente positiva. Suponemos que se dispone de una función de la clase SuministroDiario *cuantos(i)* que devuelve el número de tartas que quedan de tipo i.

Ejemplo: supongamos que hay cuatro tipos de tarta, 1, 2, 3 y 4, y quedan 0, 2, 5 y 0 tartas de cada tipo respectivamente. Si a la máquina le tocase fabricar una tarta de tipo 1 ó 4, como no quedan provisiones de esos tipos de tarta tendría que fabricar una tarta de tipo 2. Si a la máquina le tocase fabricar una tarta de tipo 2, como si hay suministros para fabricar tartas de ese tipo, en caso de poder fabricarla (si se cumplen las condiciones necesarias) al final quedarían suministros para fabricar 0, 1, 5 y 0 tartas de cada tipo (por orden), y si le hubiese tocado fabricar una de tipo 3 quedarían 0, 2, 4 y 0 tartas de cada tipo (por orden).

1. Se define la función de Fibonacci como:

$$f(0) = f(1) = 1$$
$$f(n) = f(n-1) + f(n-2)$$

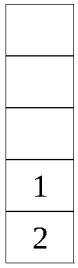
a) (0,5 puntos). Realizar una implementación recursiva de la función de Fibonacci siguiendo directamente su definición.
b) (0,75 puntos). Calcular el coste asintótico temporal de la función implementada.
c) (0,75 puntos). Una llamada recursiva se considera abierta si se ha realizado la llamada, pero aún no se ha devuelto un resultado. Calcule, en función de su parámetro de entrada, el número máximo de llamadas recursivas abiertas para la función de Fibonacci implementada.

2. Se dice que una lista es homogénea si todos sus elementos se repiten el mismo número de veces. Por ejemplo, la lista [3,2,3,1,1,2] es homogénea porque todos sus elementos se repiten dos veces, mientras que la lista [1,2,1] no lo es, porque el 1 se repite 2 veces y el 2 sólo 1 vez. Se pide:

a) (3 puntos). Implemente una función `boolean esHomogenea()` dentro del tipo de datos `ListIF<Integer>` que determine si la lista es homogénea.
b) (1 punto). Calcule, razonadamente, el coste asintótico temporal de la función esHomogenea() implementada en el punto anterior.

3. Una máquina pila es un sistema computacional que permite ejecutar una serie de instrucciones aritméticas utilizando una pila como memoria. A la hora de ejecutar una operación con dos argumentos, éstos son desapilados de la memoria (teniendo en cuenta el orden en el que se hubieran apilado) y, tras realizar la operación, el resultado se apila nuevamente en la memoria. Por ejemplo, si el estado de la memoria es el indicado en la pila 1 y se ejecuta una resta, el resultado tras dicha operación sería el indicado en la pila 2:

Pila 1



Pila 2

puesto que el resultado de restar 6 – 5 es 1. Se pide:

a) (1 punto) Implementar una función **`StackIF<Integer>`** `resta()` dentro del tipo **`StackIF<Integer>`**, que ejecute una operación de resta sobre la pila. En caso de que la operación no pudiera realizarse (por haber menos de dos operandos en la pila), la función deberá devolver una pila vacía.
b) (1 punto). Supongamos, además, la existencia de una función **`StackIF<Integer>`** `suma()` que ejecuta una operación de suma en las mismas condiciones que la función `resta()` del apartado anterior. Implementar una función **`StackIF<Integer>`** `ejecuta(`**`ListIF<Character>`** `L)` dentro del tipo **`StackIF<Integer>`**, que devuelva el estado de la pila tras ejecutar sucesivamente una lista de operaciones de sumas y restas codificadas en forma de los caracteres `'+'` y `'-'` respectivamente.

A continuación se encuentran los interfaces de los TAD estudiados en la asignatura a modo de apoyo para la realización del examen.

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 un 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 (Arbol 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 el arbol
       contiene el elemento
     * @param elem Elemento
       buscado */
    public boolean contains (T
       element);
    /* Devuelve: un iterador para
       el arbol
     * @param traversalType el
       tipo de recorrido, que
     * sera PREORDER, POSTORDER o
       BREADTH */
    public IteratorIF<T>
       getIterator (int
       traversalType);
}
```

BTreeIF (Arbol 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 ();
}
```