Estrategias de Programación y Estructuras de Datos
Septiembre 2013 – Reserva Unión Europea

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

P1. Práctica (2 puntos). Supongamos que puede haber clientes de la pastelería que han hecho reserva previamente, de manera que serán atendidos antes que aquellos que no hayan hecho reserva. Damos por hecho que la clase Cliente se ha modificado de manera que tiene un nuevo atributo booleano "reserva" que vale *true* si el cliente hizo reserva y *false* si no la hizo y tiene una nueva función *tieneReserva()* que devuelve el valor de dicho atributo.

Teniendo en cuenta lo anterior, se pide implementar por completo la clase que ordena a los clientes según llegan. Téngase en cuenta que si llega un nuevo cliente que tiene reserva, se colocará según sea su tiempo de paciencia con respecto al resto de clientes con reserva que ya estén en el local. Si llega un nuevo cliente sin reserva, se colocará de la manera habitual, pero siempre por detrás de aquellos clientes que hayan hecho reserva.

1. Considérense dos formas de cortar una patata en tiras de $1cm^2$ de grosor: en la primera se corta la patata en rodajas de 1 cm de grosor, y después cada rodaja en tiras. En la segunda, se corta la patata en rodajas de 1 cm de grosor sosteniéndolas para que no se separen, y después se vuelve a cortar de forma transversal a los cortes iniciales. Supongamos que las patatas son cubos de N centímetros de lado y que en el primer procedimiento cada corte lleva un tiempo promedio de medio segundo, mientras que en el segundo, cada corte lleva un tiempo promedio de un segundo. Se pide:

a) (0.5 puntos). Fórmula T(N) para calcular el tiempo de cortar una patata de N centímetros de lado en tiras según cada uno de los procedimientos.
b) (0.5 puntos). Coste asintótico temporal O() de cada uno de los procedimientos
c) (0.5 puntos). Tamaño umbral que separa los N para los que un procedimiento es más eficiente y los N para los que es el otro. ¿En qué caso es útil el coste asintótico temporal? ¿Qué conclusiones podemos sacar sobre la utilidad del coste asintótico temporal?

2. (1.5 puntos). Estimar, de la forma más ajustada posible, el coste asintótico temporal del siguiente programa (atención, multiplicar los tamaños de los bucles anidados puede dar lugar a una sobreestimación del coste):

```
for( int i = 1; i <= n; i++ )
  for( int j = 1; j <= i * i; j++ )
    if( j % i == 0 )
      for( int k = 0; k < j; k++ )
        sum++
```

3. (2.5 puntos). Implementar un método `int secIguales(int k)` dentro del tipo **ListIF<Integer>** que encuentre el tamaño de la secuencia más larga dentro de la lista de enteros iguales a `k`. Por ejemplo, aplicar el método `secIguales(2)` a la lista `[1,2,2,1,4,5,3,2,2,2,4]` devolvería 3, pues hay dos secuencias de doses, una de ellas de tamaño 3.

4. (2.5 puntos). Implementar un método **`ListIF<Integer>`** `noRepes()` dentro del tipo **`ListIF<Integer>`**, que devuelva una lista eliminando los elementos repetidos que haya en la original. Por ejemplo, aplicar el método `noRepes()` a la lista `[1,2,4,5,3,3,4,2,6,1,7,2]` devolvería la lista `[1,2,4,5,3,6,7]`.

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 ();
}
```