

Estrategias de Programación y Estructuras de Datos
Junio 2013 – 1ª Semana

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

P1. Práctica: Supongamos que las operaciones que implican el transcurso de unidades de tiempo afectan a todos los clientes presentes en la pastelería. Teniendo en cuenta este cambio, se pide:

a) (1 punto). Codificar una función que modifique la paciencia de todos los clientes presentes en el local

b) (1 punto). Modificar adecuadamente el código de atención del cliente más impaciente y/o funciones auxiliares oportunas, de manera que se evite la atención de clientes sin paciencia y se modifique la paciencia de todos los clientes presentes en el local cuando sea pertinente.

1. (1,5 puntos). Discutir las limitaciones del análisis de la eficiencia de los algoritmos basado en $O()$

2. (2 puntos). Dada una lista de N enteros entre 0 y 100 (con posibles repeticiones), descríbase una forma de ordenarlos en tiempo $O(N)$. Pista: utilice un array como estructura de datos auxiliar.

3. (1,5 puntos). Prográbase el método `int numLeafs()` dentro del tipo `TreeIF` que calcula el número de hojas que tiene un árbol. Nota: no se permite el uso de iteradores.

4. (3 puntos). Implementar un método `StackIF<Integer> noRepes()` dentro del tipo `StackIF<Integer>`, que devuelva una pila eliminando los elementos repetidos que haya en la original. Por ejemplo:

Entrada:

2
3
3
2
1

Salida:

3
2
1

Pista: utilice las estructuras de datos auxiliares que considere oportunas.

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)

```
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 el el primer elemento
   * @param e2 el segundo elemento
   */
public int compare (T el, T e2);
/* Devuelve: cierto si un
   elemento es menor que otro
   * @param el el primer elemento
   * @param e2 el segundo elemento
   */
public boolean isLess (T el, T
    e2);
/* Devuelve: cierto si un
   elemento es igual que otro
   * @param el el primer elemento
   * @param e2 el segundo elemento
   */
public boolean isEqual (T el, T
    e2);
/* Devuelve: cierto si un
   elemento es mayor que otro
   * @param el el primer elemento
   * @param e2 el segundo elemento*/
public boolean isGreater (T el,
    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 ();
}

```