Primer Parcial
Grado en Ingeniería Informática, 1º,
Curso 2016-2017
Estructura de Datos y Algoritmos,
Grupo 81M, 8 de Marzo de 2017

Nombre y Apellidos:

Problema 1 (0.5 puntos). Recursión:
Escribe un método recursivo que calcule la suma de todos los números impares de x hasta 1 (ambos incluidos).

Solución:

```
/**
 * recursive method to calculate
 * the sum of all odd numbers from n until 1
 * */
public static int sumOdd(int n) {
    if (n<=0) return 0;
    else if (n==1) return 1;
    else if (n%2!=0) return n+sumOdd(n-2);
    else return sumOdd(n-1);
}
```

Problema 2 (1.5 puntos). TAD Lineales y Complejidad

Un matemático (un tanto peculiar) ha definido una nueva estructura de datos lineal, CrazyQueue, para almacenar números enteros. La especificación informal de dicha estructura es la siguiente:

Las operaciones del TAD CrazyQueue son las siguientes:

- *enqueue(entero x):* que recibe un entero x, y lo almacena en la penúltimo posición de la CrazyQueue. Es decir, justo antes del último elemento. Si la colaloca está vacía, se añadirá sin más.
- *dequeue()*: este método no elimina (y devuelve) el primer elemento almacenado en la CrazyQueue, sino que elimina (y devuelve) el siguiente, es decir, el segundo. Si la cola está vacía no se hace nada. Si la cola sólo tiene un único nodo, se borra dicho nodo y se devuelve su valor almacenado.
- *getSize()*: devuelve el número de elementos en la CrazyQueue.
- *isEmpty()*: devuelve cierto si la CrazyQueue está vacía y falso en otro caso.

Se pide:

i) (0.2) Escribe la especificación formal del TAD CrazyQueue, ICrazyQueue.

Solución:

```
public interface ICrazyQueue {

    public boolean isEmpty();

    public void enqueue(Integer elem);

    public Integer dequeue();

    public int getSize();

}
```

ii) (0.9) Dada las siguientes clases:

```
public class SNode {
Integer elem;
SNode sig;
        public SNode(Integer x) {
                elem=x;
        }
   }
```

```
public class CrazyQueue implements ICrazyQueue {
    SNode first; //primer nodo
    SNode last;    //ultimo nodo

    ....
}
```

Completa su la clase CrazyQueue, añadiendo la implementación de los métodos enqueue y dequeue. Nota: no hace falta que implementes los nodos getSize() y isEmpty().

Razona qué beneficios o inconvenientes, en términos de complejidad temporal, aporta el uso del atributo **last**, que guarda la referencia al último nodo de la CrazyQueue, respecto a no usarlo en su implementación.

Solución:

```java
/**
 * This method does not add the new node behind last, but ahead of last
 * That is, it adds it at the penultimate position
 */
public void enqueue(Integer elem) {
    SNode newNode = new SNode(elem);

    SNode prev=null;
    for (SNode nodeIt=first;nodeIt!=last;nodeIt=nodeIt.next) {
        prev=nodeIt;
    }
    //now nodeIt must be last, and prev the previous node to last

    if (isEmpty()) {
        first=newNode;
        last=newNode;
    } else {
        if (prev==null) { //the list only has a node
            newNode.next=first;
            first=newNode;
        } else {
            prev.next=newNode;
            newNode.next=last;
        }
    }
    size++;

}
```

En una implementación usando nodos simples, la complejidad temporal del método **enqueue** siempre sera líneal (es decir, O(n)), independientemente de si tenemos un atributo que guarde la referencia al ultimo nodo o que no tengamos dicha referencia, ya que de todas formas, siempre es necesario recorrer la lsita para obtener la referencia al penúltimo elemento.
El método enqueue del TAD Queue estudiada en clase, sí que se beneficia del uso del atributo last. Si no tenemos dicho atributo, es necesario recorrer toda la cola hasta llegar al ultimo nodo (aquél cuya referencia next es null), para poder añadir un nuevo element, por tanto, la complejidad es lineal (es decir, O(n)). Con el atributo last, la complejidad es constant (O(1)).

```java
public Integer dequeue() {
    if (isEmpty()) {
        System.out.println("Queue is empty!");
        return null;
    } else {
        SNode nodeToLeave=null;
        if (size==1) {
            nodeToLeave = first;
            first=null;
            last=null;
        } else {
            nodeToLeave=first.next;
            if (nodeToLeave==last) {
                last=first;
            } else {
                first.next=nodeToLeave.next;
            }
        }
        size--;

        return nodeToLeave.elem;
    }
}
```

El método dequeue() siempre tampoco se beneficia del uso del atributo last (en terminos de complejidad) ya que el número de instrucciones para borrar el primer elemento de la estructura siempre es constant, y por tanto, la complejidad es O(1).

iii) (0.4) Ahora te pedimos que implementes la estructura, pero utilizando nodos dobles. Esta vez sólo vamos a pedir que implementes el método enqueue() (es decir, no debes implementar el resto de métodos). Como condición, te pedimos que en esta implementación no uses nodos centinelas.

```
public class DNode {
Integer elem;
        DNode sig;
        DNode ant;
        public DNode(Integer x) {
                elem=x;
        }
}
public class DColaLoca implementes IColaLoca {
```

```
    DNode first; //primer nodo, ojo no es un nodo centinela!!!
    DNode last;     //ultimo nodo, ojo no es un nodo centinela!!!

    ....
}
```

Solución:

```
/**
 * This method does not add the new node behind last, but ahead of last
 * That is, it adds it at the penultimate position
 */
public void enqueue(Integer elem) {
    DNode newNode = new DNode(elem);
    if (isEmpty()) {
        first=newNode;
        last=newNode;
    } else {
        if (first==last) {
            newNode.next=last;
            last.prev=newNode;
            first=newNode;
        } else {
            DNode prevNode=last.prev;
            newNode.next=last;
            newNode.prev=prevNode;
            prevNode.next=newNode;
            last.prev=newNode;
        }
    }
    size++;
}
```