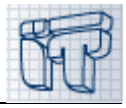


NOMBRE:
APELLIDOS:
NIA:
GRUPO:

Segundo parcial

2ª Parte: Problemas (7 puntos sobre 10)

Duración: 80 minutos
Puntuación máxima: 7 puntos
Fecha: 4 mayo 2018

Instrucciones para el examen:

- No se permite el uso de libros o apuntes, ni tener teléfonos móviles u otros dispositivos electrónicos encendidos. Incumplir cualquiera de estas normas puede ser motivo de expulsión inmediata del examen.
- Rellena tus datos personales antes de comenzar a realizar el examen.

Problema 1 (3 / 7 puntos)

Carnicerías Pelayo requiere implantar en sus locales un sistema de turnos para organizar la atención a sus clientes. El objetivo del sistema es que se atienda a los clientes en base a su orden de llegada cumpliendo los siguientes requisitos:

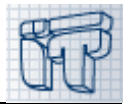
1. Existen dos tipos de clientes:
    a. Clientes *físicos* que van al local y realizan el pedido in situ.
    b. Clientes *online* que han realizado su compra por internet y vienen únicamente a recogerlo.
2. Los clientes *online* son atendidos con mayor prioridad que los *físicos*, es decir, que si llega un cliente *online* al local es el primero que es atendido tras terminar con el cliente que se esté atendiendo en ese momento.
3. Los clientes, dentro de su tipología (*físico* u *online*), serán atendidos por estricto orden de llegada y siempre priorizando los clientes *online*.
4. Un cliente del mismo tipo no puede ser atendido antes de otro que haya llegado con anterioridad.
5. Un cliente puede salir en cualquier momento del local sin realizar la compra.

Se dispone para la implementación del sistema de las clases `Node<E>`,`LinkedQueue<E>`,`Customer` y `YourTurn`.
Nota: no se muestra la implementación de todos los métodos por simplicidad.

Apartado 1 (1,5 puntos)

Se pide implementar el método `removeItem` que elimina de la cola (FIFO) el elemento que se pasa por parámetro (`E item`). Si el elemento no se encuentra en la cola no se borra ningún elemento de la misma.

```
public class Node<E> {
       private E info;
       private Node<E> next;
       public Node(){…}
       public Node(E info){…}
       public Node(E info, Node<E> next){…}
       public Node<E> getNext() {…}
       public void setNext(Node<E> next) {…}
       public E getInfo() {…}
       public void setInfo(E info) {…}
}

public class LinkedQueue<E> implements Queue<E> {
       private Node<E> top;
       private Node<E> tail;
       private int size;
       public boolean isEmpty() {...}
       public int size() {...}
       public E front() {...}
       public void enqueue(E item) {...}
       public E dequeue() {...}
```



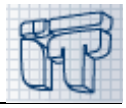
```
	public void removeItem(E item) {
		// Completar implementación
	}
}
```

Apartado 2 (1,5 puntos)

Se pide implementar de la clase `YourTurn` los siguientes métodos:

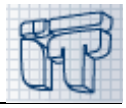
1. `void addNewCustomer(Customer customer)`: añade un nuevo cliente a la lista de clientes por atender.
2. `String getNextCustomer()`: siguiendo los criterios de priorización definidos devuelve el nombre del siguiente cliente que se debe atender y lo elimina de la lista de clientes por atender. En caso de que no haya clientes en el local, devuelve `null`.
3. `void removeCustomer(Customer customer)`: elimina el cliente que se le pasa por parámetro de la lista de clientes por atender (cliente que sale del local sin realizar la compra). Si no se encuentra el cliente lógicamente no se borra de la lista. *Nota: un cliente puede estar situado en cualquier posición de la lista y no puede haber clientes duplicados.*

```
public class Customer {

	public static final int NORMAL_CUSTOMER = 1;
	public static final int ONLINE_CUSTOMER = 2;

	private String name;
	private int type;

	public Customer(String name, int type) {
		this.name = name;
		this.type = type;
	}

	public String getName() { return name;}
	public int getType() { return type;}

	public boolean equals(Object object) {
		Customer customer = (Customer)object;
		return (customer.getName().equals(name) &&
			(customer.getType() == type));
	}
}

public class YourTurn {

	private LinkedQueue<Customer> normalQueue;
	private LinkedQueue<Customer> onlineQueue;

	public YourTurn() {
		normalQueue =  new LinkedQueue<>();
		onlineQueue = new LinkedQueue<>();
	}

	public void addNewCustomer(Customer customer) {
		// Completar implementación
	}
	public void removeCustomer(Customer customer) {
		// Completar implementación
	}
	public String getNextCustomer() {
		// Completar implementación
	}
}
```



```
public void removeItem(E item) {

  if (!isEmpty()) {
    boolean exit = false;
    Node<E> currentNode = top;

    if (currentNode.getInfo().equals(item)) {
      top = currentNode.getNext();
        exit = true;
        size--;
    }

    while(!exit) {
      Node<E> nextNode = currentNode.getNext();
        if (nextNode == null) {
        exit = true;
        } else if (nextNode.getInfo().equals(item)) {
        currentNode.setNext(nextNode.getNext());
        if (nextNode.getNext() == null) {
          tail = currentNode;
        }
        size--;
        exit = true;
      }
      currentNode = nextNode;
    }

    if (isEmpty()) {
      tail = null;
    }
  }
}
```

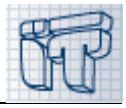
```
public class YourTurn {

  private LinkedQueue<Customer> normalQueue;
  private LinkedQueue<Customer> onlineQueue;

  public YourTurn() {
       normalQueue =  new LinkedQueue<Customer>();
       onlineQueue = new LinkedQueue<Customer>();
  }

  public void addNewCustomer(Customer customer) {
       if (customer.getType() == Customer.NORMAL_CUSTOMER) {
       normalQueue.enqueue(customer);
       } else if (customer.getType() == Customer.ONLINE_CUSTOMER) {
       onlineQueue.enqueue(customer);
       }
  }

  public String getNextCustomer() {
       String customerName = null;
       if (!onlineQueue.isEmpty()) {
       customerName = normalQueue.dequeue().getName();
       } else if (!normalQueue.isEmpty()) {
       customerName = onlineQueue.dequeue().getName();
       }
       return customerName;
  }

  public void removeCustomer(Customer customer) {
       if (customer.getType() == Customer.NORMAL_CUSTOMER) {
       normalQueue.removeItem(customer);
       } else if (customer.getType() == Customer.ONLINE_CUSTOMER){
       onlineQueue.removeItem(customer);
       }
  }
}
```



Problema 2 (3 / 7 puntos)

La clase `IBSTree` representa un árbol binario de búsqueda en el que la clave de búsqueda es la propia información guardada, de tipo `Integer`. Dado el siguiente esqueleto, se pide completar los métodos de la clase `IBSTree`.

```
public class IBSTree {

	private IBSNode root;

	class IBSNode {

		private Integer data;
		private IBSTree left;
		private IBSTree right;

		IBSNode(Integer data, IBSTree left, IBSTree right) {
			this.data = data;
			this.left = left;
			this.right = right;
		}
	}

	public IBSTree() { root = null; }

	public IBSTree(Integer data) {
		// completar (aptdo. 1)
	}

	public Integer getData() { return root.data; }
	public void setData(Integer d) { this.root.data = d; }

	public IBSTree getLeft() { return root.left; }
	public IBSTree getRight() { return root.right; }

	public boolean isEmpty() { return root == null; }

	/**
	 * Multiplica el dato almacenado en cada nodo
	 * por el factor que se pasa como parámetro.
	 * @param factor número por el que se multiplican los datos
	 */
	public void scale(int factor) {
		// completar (aptdo. 2)
	}

	/**
	 * Devuelve el dato almacenado en el árbol
	 * más próximo al que se pasa como parámetro.
	 * Devuelve null si el árbol está vacío.
	 */
	public Integer searchClosest(Integer data) {
		return searchClosest(data, null);
	}

	/**
	 * Método auxiliar para encontrar el dato almacenado en el árbol
	 * más próximo al que se pasa como parámetro.
	 * Devuelve null si el árbol está vacío.
	 * @param data dato a buscar
	 * @param closestFound dato más próximo encontrado hasta el momento
	 * (null si no se ha encontrado nada aún)
	 */
	private Integer searchClosest(Integer data, Integer closestFound) {
		// completar (aptdo. 3)
	}
}
```



Apartado 1 (0,25 puntos)

Programa el constructor `public IBSTree(Integer data)` que inicializa un árbol con la información que se pasa como parámetro en su nodo raíz.

Puedes escoger la implementación que prefieras, inicializando los hijos a `null` o a árboles vacíos. Pero ten en cuenta que los métodos programados en los siguientes apartados deben ser coherentes con la implementación que decidas en el constructor.

Apartado 2 (1,25 puntos)

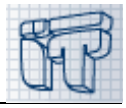
Programa el método `scale` que aplica un factor de escala a cada uno de los datos almacenados en el árbol. Es decir, multiplica el dato almacenado en cada nodo por el factor que se indica.

Apartado 3 (1,5 puntos)

Programa el método auxiliar `searchClosest` que busca el dato almacenado en el árbol más próximo al que se pasa como parámetro. El parámetro auxiliar `closestFound` representa el resultado parcial, la mejor aproximación encontrada hasta el momento.

Solución:

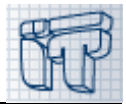
```
public class IBSTree {

	class IBSNode {
		private Integer data;
		private IBSTree left;
		private IBSTree right;
		IBSNode(Integer data, IBSTree left, IBSTree right) {
			this.data = data;
			this.left = left;
			this.right = right;
		}

	}

	private IBSNode root;

	public IBSTree() {
		root = null;
	}

	public IBSTree(Integer data) {
	   root = new IBSNode(data, null, null); // opción A
	   //root = new IBSNode(data, new IBSTree(), new IBSTree());//opc.B
	}

	public Integer getData() { return root.data; }
	public void setData(Integer d) { this.root.data = d; }

	public IBSTree getLeft() { return root.left; }
	public IBSTree getRight() { return root.right; }

	public boolean isEmpty() { return root == null; }



	// Apartado 1

	/**
	 * Multiplica el dato almacenado en cada nodo
	 * por el factor que se pasa como parámetro.
	 * @param factor número por el que se multiplican los datos
	 */
	public void scale(int factor) {
		if (!this.isEmpty()) {
			this.setData(this.getData().intValue() * factor);
			if (this.getLeft() != null) {
				this.getLeft().scale(factor);
			}
			if (this.getRight() != null) {
				this.getRight().scale(factor);
```



```
                        }
                }
        }


        // Apartado 2

        /**
         * Devuelve el dato almacenado en el árbol
         * más próximo al que se pasa como parámetro.
         * Devuelve null si el árbol está vacío.
         */
        public Integer searchClosest(Integer data) {
                return searchClosest(data, null);
        }

        /**
         * Método auxiliar para encontrar el dato almacenado en el árbol
         * más próximo al que se pasa como parámetro.
         * Devuelve null si el árbol está vacío.
         * @param data dato a buscar
         * @param closestFound dato más próximo encontrado hasta el momento
         * (null si no se ha encontrado nada aún)
         */
        private Integer searchClosest(Integer data, Integer closestFound) {
                if (this.isEmpty()) {
                        return null;
                }
                Integer res = closestFound;
                // Comparar con el nodo actual
                int dif = Math.abs(this.getData().intValue() - data.intValue());
                if (dif == 0) {
                        // Encontramos uno igual: no puede haber una aproximación mejor
                        return this.getData();
                }
                if ((closestFound == null) ||
                                (dif < Math.abs(closestFound.intValue() - data.intValue()))) {
                        // la información del nodo actual es una aproximación mejor
                        res = this.getData();
                }
                // Si no hemos encontrado el dato exacto,
                // seguimos buscando si hay una aprox mejor
                if ((data.intValue() < this.getData().intValue()) && (this.getLeft() != null)) {
                        // si el dato a buscar es menor que el actual, busca por la izquerda
                        return this.getLeft().searchClosest(data, res);
                } else if (data.intValue() > this.getData().intValue()) {
                        // si el dato a buscar es mayor que el actual, busca por la derecha
                        return this.getRight().searchClosest(data,res);
                }
                return res;

        }

}
```

Problema 3 (1 / 7 puntos)

Dado el array `int a[]`, programa el algoritmo de ordenación (criterio: menor a mayor) qué más intercambios de posición realiza de todos los que se han visto en la asignatura.



Solución

Bubble Sort.

```
public static void bubbleSort (int[] a) {
  for (int i=0; i<a.length-1; i++) {
    for (int j=0; j<a.length-1-i; j++) {
      if (a[j]>a[j+1]){
            swap(a, j, j+1);
      }
    }
  }
}

public static void swap (int[] a, int i, int j) {
        int aux=a[i];
        a[i]=a[j];
        a[j]=aux;
}
```