Grado en Ingeniería Informática y Doble Grado en Informática y Administración de Empresas	**Asignatura Estructura de Datos y Algoritmos** **24 de Marzo de 2014.** **SEGUNDO EXAMEN PARCIAL**

Nombre:

.....

Apellidos:

.....

Grupo: .....

LEA ATENTAMENTE ESTAS INSTRUCCIONES ANTES DE COMENZAR LA PRUEBA

1. Es necesario poner todos los datos del alumno en el cuadernillo de preguntas (este documento). Use un bolígrafo para rellenarlos.
2. El examen de la parte de teoría está compuesto por 6 Preguntas.
3. Solamente se evaluará la contestación en este cuadernillo de preguntas.
4. Cuando finalice la prueba, se deben entregar el enunciado del examen y cualquier hoja que haya empleado.
5. No está permitido salir del aula por ningún motivo hasta la finalización del examen.
6. Desconecten los móviles durante el examen.
7. La duración del examen es de **2 horas**.
8. **Todos los métodos necesarios de las librerías son públicos y los atributos de los TADs son públicos.**

NO PASE DE ESTA HOJA hasta que se le indique el comienzo del examen

1. Definir el TAD lineal necesario para poder gestionar las peticiones de envío de paquetes que llegan a un Centro Logístico. Las peticiones se almacenarán por orden de llegada al Centro Logístico. Cuando una petición alcanza la primera posición del TAD, la petición pasa a procesarse para su envío (solo puede salir del TAD por esa posición) **(TOTAL 2,25 pto.)**

a) Definir en Java el TAD que mejor se ajuste para resolver el problema teniendo en cuenta que todos las peticiones son homogéneas de tipo String **(0,25 pto).**

b) Cada cierto tiempo es necesario recolocar **una de las peticiones** que esperan en función del siguiente procedimiento. Teniendo en cuenta que la posición más cercana a Centro Logístico es 0, la petición k que se encuentra en la posición k-1 dentro del TAD pasa a estar la primera para disminuir su tiempo de espera. En ningún caso debe perderse una petición y el resto de peticiones deben mantener su orden.

Implementar el método recolocar() **(2 pto.)**

Observaciones:

Solo pueden usarse los métodos propios del TAD utilizado para recorrer la estructura.

Se pueden utilizar otros TAD auxiliares simples o complejos (Pila, Cola, Listas, Nodos, etc.) y no existe el método size para los TADs a utilizar.

Por ejemplo:

(posición 0) *(posición 4)*

CENTRO ←[pet98, pet8, pet12, pet57, pet32, pet111, pet42]

Reordenamos para K=5

CENTRO ← [**pet32**, pet98, pet8, pet12, pet57, pet111, pet42]

a) Teniendo en cuenta que es necesario mantener el orden de llegada de las peticiones el TAD que mejor se ajusta al problema es el TAD Cola.

b) Se elige como TAD auxiliar un TAD Lista Simple. Otras soluciones validas incluyen el uso de un TAD Cola.

```java
package edalib.list.singlelink;
   // Completar definición de clase para definir el TAD necesario
public class TADPeticiones extends SQueue<String>

   /* Implementar Método reordenar */
      public void reordenar(int k) {
            //Contador para recorrer los paquetes
            int contador=0;

            //lista auxiliar
            SList <String> lista = new SList<String>();

            //Nodo para el elemento k+1
            String elem="";

            while (!isEmpty()) {
                  //metemos en la pila los elementos de la cola
                  if (contador!=k-1){lista.addLast(dequeue());}
                  else{
                        elem=dequeue();

                  }
                  contador++;
            }

            System.out.println(toString());

            //el primer elemento de la pila debería ser el nuevo front
            System.out.println("El elemento a recolocar es " +elem);


            //Nodo para recorrer la  lista
            SNode<String> nodo = lista.getFirstNode();

            //el primer nodo tiene que ser elem
            enqueue(elem);

            while (nodo!=null) {
                  //el resto de elementos de la pila se meten en la cola
                        enqueue(nodo.getElement());
                        nodo = nodo.getNextNode();
            }
      }
}
```

2. Implementar el código del método toString() del TAD Pila empleando únicamente las operaciones push() y pop() vistas en clase. Al terminar el método el TAD Pila tiene que quedar en el formato original. (**TOTAL 1,5 pto.)**

El formato de salida de la Pila por pantalla tendrá que tener este formato:

$$(Elemento_1, Elemento_2, \ldots\ldots, Elemento_n)$$

Nota: Se pueden utilizar otros TAD auxiliares (Pila, Cola, Listas, etc.) y no existe el método size para los TADs a utilizar.

```
public String toString() {
		String result = null;


		//Creamos una pila auxiliar

		SStack<E> pilaAux = new SStack<E>();

		//recorremos la pila y desapilamos todos los elementos
		while(!isEmpty())
		{
			if (result == null) {
				result = "(" + top().toString();
			} else {
				result += "," + top().toString();
			}

			//desapilamos y apilamos en la pila aux
			pilaAux.push(this.pop());
		}


		while(!pilaAux.isEmpty()){
			push(pilaAux.pop());

		}

		return result == null?"empty":result+")";
	}
```

3. Explicar brevemente el funcionamiento del siguiente código asociado a un método del TAD Lista Doblemente Enlazada. Comentar los posibles errores detectados.

Puede incluir un diagrama explicativo si se considera oportuno (**TOTAL 0,5 pto)**

```
public void desconocido(DNode<E> newNode, index int) {
  DNode<E> newNode = new DNode<E>(elem);
  int i = 0;

  DNode<E> nodeIt = header;
       while (nodeIt != tailer) {
              if (i == index+1) {
                     newNode.nextNode = nodeIt.nextNode;
                     newNode.previousNode = nodeIt;
                     nodeIt.nextNode.previousNode = newNode;
                     nodeIt.nextNode = newNode;
                     return;
              }
              ++i;
       }
       System.out.println("Índice fuera de límites");
}
```

Método que recorre la lista doble desde la cabecera hasta el final usando el nodo auxiliar nodeIt y cuando llega a la posición index+1 inserta el nuevo nodo NewNode en esa posición.

Errores: Se ha duplicado el nodo newNode.
Falta por definir la variable index.
El nodo nodeIt no se mueve nunca (falta nodeIt = nodeIt.nextNode).
La variable index no ha sido definida.

Implementación correcta:

```
       public void desconocido(int index, E elem) {
              DNode<E> newNode = new DNode<E>(elem);

              int i = 0;
              DNode<E> nodeIt = header;

              while (nodeIt != tailer) {
                     if (i == index) {
                            newNode.nextNode = nodeIt.nextNode;
                            newNode.previousNode = nodeIt;
                            nodeIt.nextNode.previousNode = newNode;

                            nodeIt.nextNode = newNode;
                            return;
                     }
                     nodeIt = nodeIt.nextNode;
                     ++i;
              }
              System.out.println("DList: Insertion out of bounds");
       }
```

4. Dado el siguiente código de la Clase Inversa que invierte una palabra que se introduce por teclado **(TOTAL 1,5 pto)**:

c) Completar el constructor Inversa para añadir en el TAD Lista Doble los caracteres de una palabra dada por parámetro **(0,25 pto).**

d) Implementar el método invertir que invierte la palabra almacenada en el TAD Lista Doble y la devuelve en un String. Este método no imprime nada por pantalla. **(1 pto.)**

e) Completar el método main de la clase Inversa para realizar la llamada al método invertir y mostrar por pantalla el resultado **(0,25 pto).**

Nota: Se pueden utilizar otros TAD auxiliares (Pila, Cola, Listas, etc.) y no existe el método size para los TADs a utilizar.

```
package edalib.list.doublelink;
import java.util.Scanner;

public class Inversa extends DList<Character> {

	Inversa(String cadena){
		if (cadena==null || cadena.length()==0){
			System.out.println("La palabra introducida es vacía");
		}
		char vectorCaracteres[] = cadena.toCharArray();

		int i=0;

		while (i<vectorCaracteres.length){
			/* Completar Método */
			addLast(vectorCaracteres[i]);

			i++;
		}
	}


	public String invertir(){
		/* Implementar Método */

		String cadena="";
		DNode<Character> nodoDcho = tailer.previousNode;

		while(nodoDcho!=header ){

			cadena +=nodoDcho.getElement();

			nodoDcho = nodoDcho.previousNode;

		}


		return cadena;
```

```
	}


	public static void main(String[] args) {
		System.out.println("Introduce una palabra: ");
		Scanner objScan = new Scanner(System.in);
		String pal = objScan.nextLine();
		objScan.close();

		/* Completar Método */
		Inversa inversa = new Inversa(pal);
		String palinv = inversa.invertir();
		System.out.println(palinv);

	}

}
```

5. Implementar un método concatenarMitad() en Java que reciba por parámetro dos listas simples (listaSimple1 y listaSimple2) y que devuelve una lista Simple (listaSalida) que incluye la mitad inferior de los nodos de listaSimple1 y la mitad superior de los nodos de listaSimple2 **(TOTAL 2,5 pto)**.

Por ejemplo:

listaSimple1: **N1 N2 N3 N4**

listaSimple2: **M1 M2 M3 M4 M5 M6 M7**

listaSalida:

N1 N2 M5 M6 M7

Se puede recurrir al método getSize() que devuelve el tamaño de una lista Simple.

a) Implementar el método concatenarMitad **(1,5 pto).**

Suponemos listas de Enteros. Cual otro tipo de datos válido se ha considerado correcto para definir los TADs (String, Character, etc.)

```
   /* Implementar Método concatenarMitad() */
public static SList<Integer> ConcatenarMitad (SList<Integer> lista1,
SList<Integer> lista2 ){

		SList<Integer> listaSalida = new SList<Integer>();
		SNode<Integer> nodoAux = lista1.firstNode;
		int i =0;

		//recorremos la mitad de nodos de lista1
		while (nodoAux!=null & i<lista1.getSize()/2 ) {
			listaSalida.addLast(nodoAux.getElement());
			nodoAux=nodoAux.nextNode;
			i++;
```

```
            }

            nodoAux = lista2.firstNode;
            i =0;
            //recorremos los nodos de lista 2
            while (nodoAux!=null ) {
                  //solo insertamos los nodos que estan en la mitad superior
de la lista
                  if (i>lista2.getSize()/2 )
listaSalida.addLast(nodoAux.getElement());
                  nodoAux=nodoAux.nextNode;
                  i++;
            }

            return listaSalida;
      }
```

a) Calcule el tiempo de ejecución del algoritmo **(0,25 pto).**

T(n) = cn + x (donde c y x son constantes numéricas).

b) Estime el orden del algoritmo O(n) en cuanto a su complejidad temporal **(0,5 pto)**

O(n)

c) Teniendo en cuenta la tabla de jerarquías para los órdenes de Complejidad: Proporcione si es posible al menos dos órdenes de Complejidad menores y otros dos mayores respecto al del método concatenarMitad **(0,25 pto)**.

O(1) < O(logn) < O(n) < O(n2) < O(n3)

6. Implementar un método recursivo encuentra() que permita busca un determinado elemento en una TAD Cola de Enteros y devuelva la posición en la que se encuentra el elemento (si no lo encuentra devolverá el valor -1) **(TOTAL 1,75 pto)**.
    a) Crear la clase RecurCola para que sea una cola de tipo Enteros **(0,25 pto).**
    b) Implementar el método encuentra **(1,25 pto).**
    c) Completar el método main de la clase RecurCola que se facilita e incluir el diagrama o figura de la pila de llamadas que se genera al probar el método encuentra y el resultado que se obtiene (**0,25 pto).**

```
package edalib.list.singlelink;

//Completar definición clase RecurCola para incluir el TAD Cola
public class ColaRecur  extends SQueue<Integer>{


	private int encuentra(int a) {

		if (isEmpty()) {
			return -1;
		}
		if (front() == a) {
			return 0;
		}
		//	quito el elemento temporalmente
		int aux = dequeue();

		//	sigo buscando en la pila (con un elemento menos)
		int result = encuentra(a);
		if (result >=0) {
				result ++;
		}

		//vuelvo a añadir el elmento la pila
		enqueue(aux);
		return result;
}


	public static void main(String[] args) {
		ColaRecur cola = new ColaRecur();
		cola.enqueue(4);
		cola.enqueue(1);
		cola.enqueue(3);
		cola.enqueue(2);
		System.out.println(cola);
		System.out.println("El elemento está en la posición " +
cola.encuentra(2));
		System.out.println(cola.encuentra(8));
		System.out.println(cola);
	}

}
```