ESTRUCTURA DE DATOS Y ALGORITMOS **2-Septiembre-2008**
Titulación: Ingeniero Técnico en Informática de Gestión **Curso: 2º**

NOTA FINAL: Nota Practica (1 punto) + Nota Examen (9 punto)

- Es indispensable aprobar el examen (4,5 puntos) para sumar la puntuación de la práctica y para aprobar la asignatura es necesario una nota superior o igual a 5 puntos
- La práctica es opcional

1. Ejercicio (1 puntos)

A={3, 13, 8, 25, 45, 23, 98, 58}→A={3, 8, 13, 25, 45, 23, 98, 58}→ A={3, 8, 13, 25, 45, 23, 98, 58}→ A={3, 8, 13, 25, 45, 23, 98, 58}
azul: la parte ordenada, rojo: la posición seleccionada(o el siguiente valor a ordenar)
subrayado: el resultado final

2. Ejercicio (1'5 puntos)

1. opción: usando la interfaz Comparable

```
public class Persona implements Comparable {
  private String nombre;
  private int edad;

  public Persona(String nombre, int edad) {
    this.nombre = nombre;
    this.edad = edad;
  }
  // los metodos get y set de cada variable

  public int compareTo(Object o) {
    Persona p = (Persona) o;
    If (this.edad < p.edad) return -1;
    else if (this.edad == p.edad) return 0;
    return 1;
  }
}

public class Algoritmos{
 public static void selectionSort(Comparable a[]){
   for(int i=0; i < a.length-1; i++){
     Comparable current = a[i];
     int k=i;
     for(int j=i+1; j < a.length; j++) {
        if (current.compareTo(a[j]) < 0){
          k=j;
          current=a[j];
        }
     }
     a[k]=a[i];
     a[i]=current;
 }
}

public class Principal{
  public void ordenarPersonas(Persona p[]){
    Algoritmos.selectionSort(p);
  }
}
```

2. opción: usando la interfaz Comparator

```
import java.util.Comparator;
public class Persona {
  private String nombre;
  private int edad;
  public static final Comparator compararPorEdad;
 static {
  compararPorEdad = new CompararPersonaPorEdad();
 }

  public Persona(String nombre, int edad) {
    this.nombre = nombre;
    this.edad = edad;
  }
  // los metodos get y set de cada variable
}

import java.util.Comparator;
public class CompararPersonaPorEdad implements
Comparator {

  public int compare(Object o1, Object o2) {
    Persona p1 = (Persona) o1;
    Persona p2 = (Persona) o2;
    If (p1.edad < p2.edad) return -1;
    else if (p1.edad == p2.edad) return 0;
    return 1;
  }
}

public class Algoritmos{
 public static void selectionSort (Object a[],
                                   Comparator comparator) {
   for(int i=0; i < a.length-1; i++){
     Object current = a[i];
     int k=i;
     for(int j=i+1; j < a.length; j++) {
        if (comparator.compare(current, a[j]) < 0){
          k=j;
          current=a[j];
        }
     }
     a[k]=a[i];
     a[i]=current;
 }
}

public class Principal{
  public void ordenarPersonas(Persona p[]){
    Algoritmos.selectionSort(p,
                        Persona.compararPorEdad);
  }
}
```

ESTRUCTURA DE DATOS Y ALGORITMOS **2-Septiembre-2008**
Titulación: Ingeniero Técnico en Informática de Gestión **Curso: 2º**

3. Ejercicio (1 punto)

{8, 13, 17, **26**, 44, 56, 88, 97} → {8, **13**, 17}→{**17**}→no encontrado(-1)
verde: es el valor que se encuentra en la mitad del array y utilizado para la comparación

4. Ejercicio. (2'5 puntos)

a. (0,5) **Definir** las clases necesarias para definir la estructura de listas enlazadas para almacenar cualquier tipo de contenido y las variables necesarias para implementar el método que se pide.

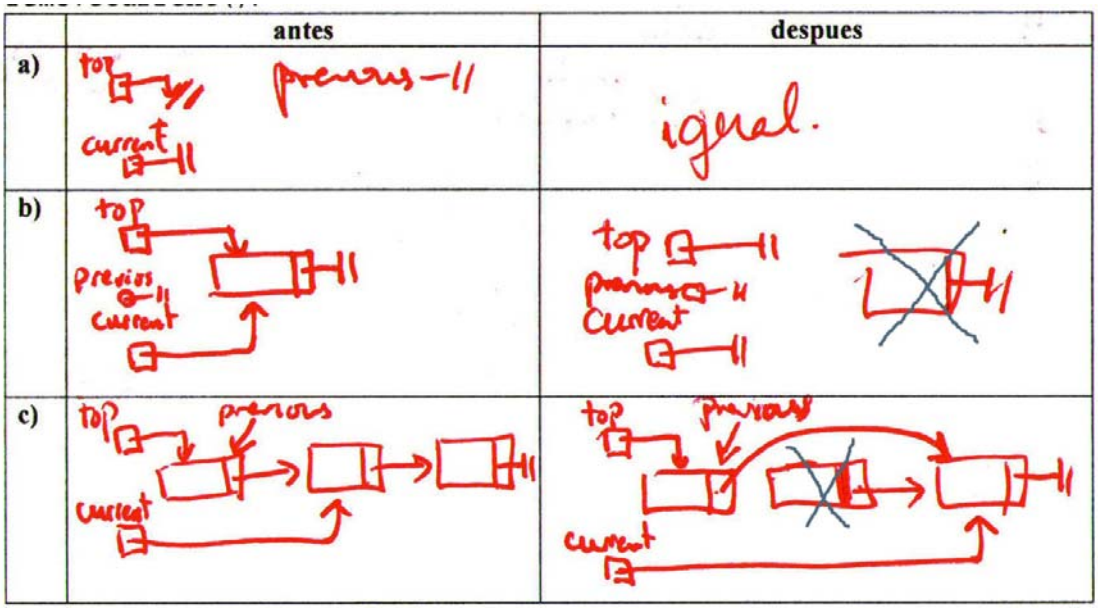
```
class Node {
   Object element;
   Node next;
}
public class LinkedList{
   Node top;
}
```

```
public class LinkedListItr {
   LinkedList theList;
   Node current;   //nodo actual
   Node previous; //nodo anterior a
                  //current
}
```

b. (1,5) **Implementar** el método `removeCurrent()`.

```
public Object removeCurrent() throw InvalidException {//implementado en la clase
                                                      //LinkedListItr
   Object obj = null;
   If (!theList.isOnList()) //caso a) Lista vacia o current no está en ningún nodo
       throws new InvalidException("No hay contenido")
   else if (theList.top == current) { //caso b) current en el primer nodo
       obj = current.element;
       theList.top = current.next;
       current = theList.top;
    }
   else { //caso c) current en el último nodo o algún otro nodo distinto al primero
      obj = current.element;
      previous.next = current.next;
      current = previous.next;
   }
   return obj;
}
```

c. (0,5) Utilizando un ejemplo o varias, **realizar** para cada caso posible dos figuras que describan una lista enlazada y el estado de las variables utilizadas **antes y después de ejecutar** el método `removeCurrent()`.



5. Ejercicio (3 puntos)

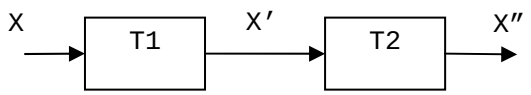
a)

Diseño del método `codificar`

Dividiremos el problema en dos pasos (transformaciones) tal como se comenta en el enunciado. Llamaremos `T1` al paso que transforma `X` en `X'` y `T2` al que obtiene `X″` dado `X'` (ver Tabla 1). El tipo de dato que utilizaremos para las entradas y salidas de las transformaciones será la cola de caracteres. Este tipo de dato nos parece adecuado para representar un mensaje, dado su lectura de izquierda a derecha, siendo el primer carácter de la cola el primer carácter de la izquierda del mensaje y el último el primero de la derecha.

Tabla 1. Diseño e implementación del método codificar

Diseño	Implementación

`codificar`(X:cola)→X":cola 	Queue **codificar**(Queue X){
	Queue X1 = T1(X) ;
	Queue X2 = T2(X1) ;
	return X2 ;
	}

A continuación pasaré a diseñar e implementar los métodos `T1` y `T2`. De aquí en adelante, para identificar el mensaje de entrada utilizaré CE (cola de entrada) y para el de salida CS (cola de salida).

1) diseño del método `T1`(CE:cola) → CS:cola

La estrategia que vamos a seguir es que cada sucesión de vocales consecutivos se moverán directamente de CE a CS, mientras que cada sucesión de caracteres consecutivos que no sean vocales se guardarán en una pila y posterior se moverán a CS. De esta forma, en el caso de la sucesión de caracteres consecutivos que no sean vocales se habrá obtenido su imagen especular (ver Tabla 2). Para ello, vamos consultando el carácter que se encuentra al frente de la cola y lo vamos moviendo a la estructura correspondiente dependiendo de si es vocal o no.

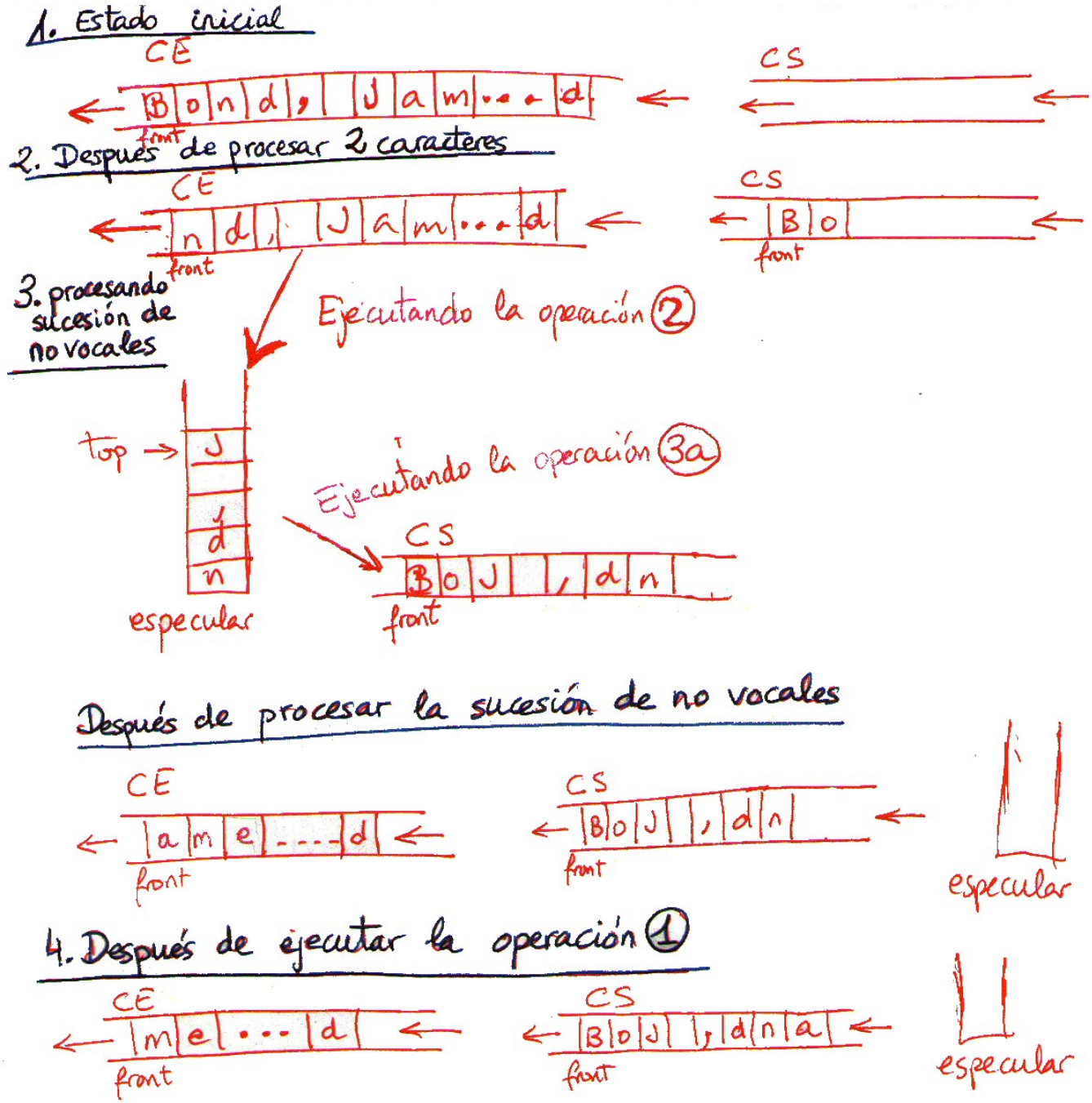
Tabla 2. Diseño e implementación del método `T1`

Algoritmo	Implementación

`T1`(CE:cola) → CS:cola
mientras haya caracteres en CE
1. mientras haya caracteres en CE y el frente de CE sea vocal
a. extraer carácter de CE y encolarlo en CS
fin mientras
2. mientras haya caracteres en CE y el frente de CE no sea vocal
a. extraer carácter de CE y apilarlo en una pila
fin mientras
3. si la pila no está vacía
a. mover todos los caracteres de la pila a CS
fin mientras
devolver CS
	Queue **T1**(Queue ce){
	Queue cs = new Queue();
	Stack especular = new Stack();
	while (!ce.isEmpty()){
	while ((!ce.isEmpty()) && (esVocal(ce.front())))
	cs.enqueue(ce.dequeue());
	while ((!ce.isEmpty()) && (!esVocal(ce.front())))
	especular.push(ce.dequeue());
	if (!especular.isEmpty())
	moverDePilaACola(especular, cs);
	}
	return cs;
	}

Para ilustrar mejor el proceso T1 vamos a utilizar una figura y describir el estado de las estructuras de datos en diferentes momentos de su ejecución (ver Figura 1).

Figura 1. Estados de las estructuras de datos que se han utilizado en la implementación de T1



En la implementación del método `T1` anterior faltan por implementar los métodos `esVocal()` y `moverDePilaACola()`. El primero de los métodos verifica si el carácter dado es una vocal o no, devolviendo true o false, respectivamente. El segundo método, en cambio, mueve todos los elementos que se encuentran en una pila a una cola (ver Tabla 3).

Tabla 3. Diseño e implementación de los métodos esVocal y moverDePilaACola.

Algoritmo	Impementación

esVocal(car:Character)→boolean
si car es "a" o "e" o "i" o "o" o "u" entonces
devolver trae
si no
devolver false
	boolean **esVocal**(Character car){
	if ((car.compareTo(new Character("a"))) ||
	(car.compareTo(new Character("e"))) || …
	(car.compareTo(new Character("u")))
	return true;
	else
	return false;
	}

moverDePilaACola(P:pila, C:cola)
mientras haya elementos en P
mover de P a C
fin mientras
	void **moverDePilaACola**(Stack p, Queue c){
	while (!p.isEmpty())
	c.enqueue(p.pop());
	}

Otra posible estrategia para resolver la transformación T1 sería el de extraer carácter por carácter del mensaje y analizar cada carácter si es vocal o no. En caso de que no sea vocal, pertenece a la sucesión y por lo tanto lo apilamos en una pila para poder obtener su imagen especular. En caso de que sea vocal y la pila esté llena significa que la sucesión de los no vocales ha terminado. Por lo tanto, primero moveremos todos los caracteres de la pila a CS y posterior encolaremos la vocal. De esta forma, seguiremos hasta que hayamos terminado de analizar todos los caracteres del mensaje de entrada (CE). Finalmente, comprobaremos si la pila está llena (en caso de que el mensaje termine con un carácter que no sea vocal) y en caso afirmativo, moveremos todos los caracteres de la pila a CS. El método devuelve CS, el mensaje de salida obtenido.

Algoritmo	Implementación

T1(CE:cola) → CS:cola
1. mientras haya caracteres en CE
a. car = extraer carácter de CE
b. si car no es vocal
apilar car en una pila
si no
mover todos los caracteres de la pila a CS
encolar car en CS
fin si
fin mientras
2. si la pila no está vacía
a. mover todos los caracteres de la pila a CS
devolver CS
	Queue **T1**(Queue ce){
	Queue cs = new Queue();
	Stack especular = new Stack();
	while (!ce.isEmpty()){
	if (!esVocal(ce.front()))
	especular.push(ce.dequeue());
	else {
	moverDePilaACola(especular, cs);
	cs.enqueue(ce.dequeue());
	}
	}
	if (!especular.isEmpty())
	moverDePilaACola(especular, cs);
	return cs;
	}

2) diseño del método T2(CE:cola) → CS:cola

La estrategia que vamos a seguir es ir moviendo el primer carácter y el último carácter de CE a CS y así sucesivamente hasta que no haya caracteres en CE (ver Tabla 4).

Tabla 4. Diseño e implementación del método T2.

Algoritmo	Impementación

T2(CE:cola)→CS:cola
mientras haya caracteres en CE
1. extraer carácter de CE y encolarlo en CS
2. si hay caracteres en CE
a. car = extraer último carácter de CE
b. encolar car en CS
fin mientras
devolver CS
	Queue **T2**(Queue ce){
	Queue cs = new Queue();
	while (!ce.isEmpty()){
	cs.enqueue(ce.dequeue());
	if (!ce.isEmpty())
	cs.enqueue(extraerUltimoDeCola(ce));
	}
	return cs;
	}

El paso 2a. requiere realizar un nuevo método, que dado una cola extraiga el último elemento de la cola (ver Tabla 5).

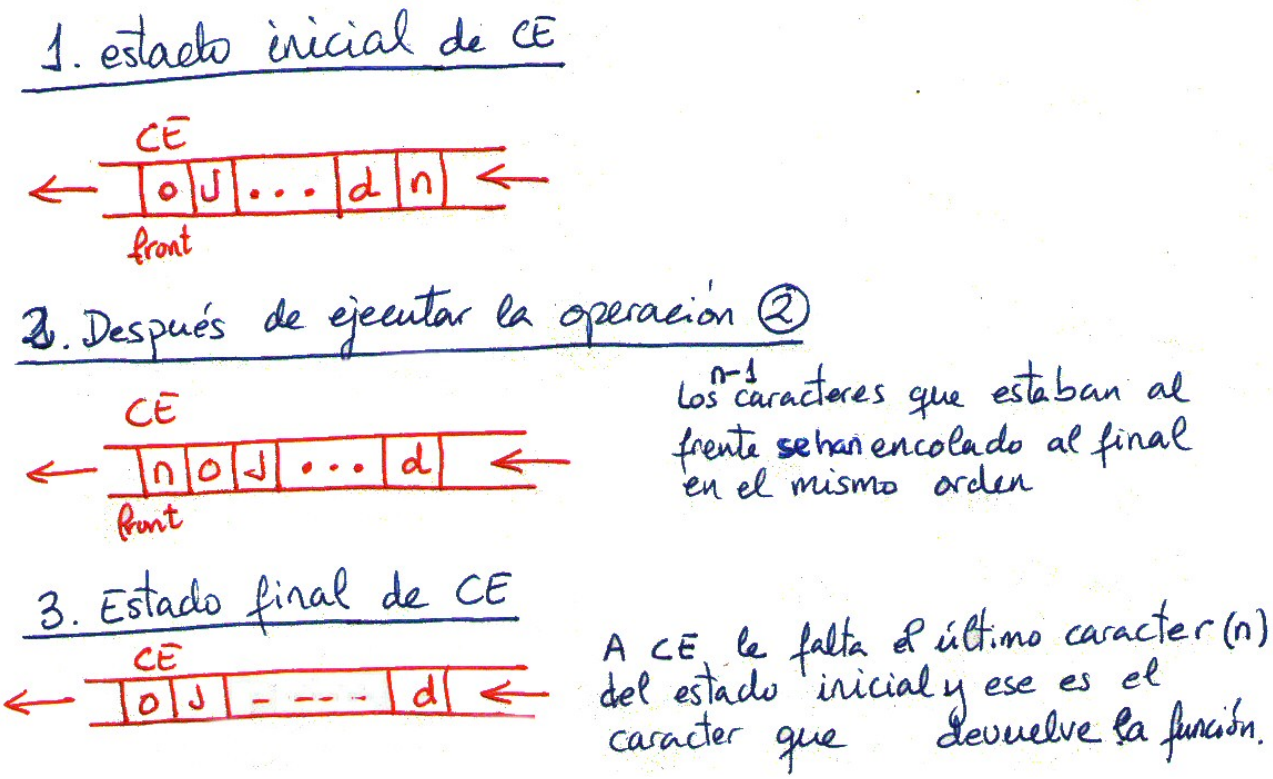
Tabla 5. Diseño e implementación del método extraerUltimoDeCola

Algoritmo	Impementación

extraerUltimoDeCola(CE:cola) → car:Character
1. n = obtener el número de carácteres en CE
2. repetir n-1 veces
a. mover el primero de CE al final de la cola
3. car = extraer carácter de la cola
4. devolver car
	Character **extraerUltimoDeCola**(Queue ce){
	int n = ce.size();
	for(int i=1;i<n;i++)
	ce.enqueue(ce.dequeue());
	return ce.dequeue();
	}

La Figura 2 describe el estado de la cola en la ejecución del método descrito arriba.

Figura 2. Estado de la cola en la ejecución del método extraerUltimoDeCola

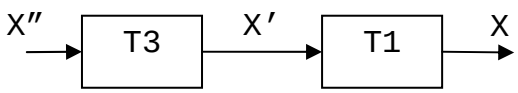


Diseño del método `descodificar`

De la misma forma que el método codificar, ésta también lo dividiremos en dos transformaciones. Llamaremos `T3` al paso que transforma `X″` en `X'` y reutilizaremos la transformación `T1` para convertir el mensaje `X'` en `X`, ya que es el mismo paso de convertir `X′` en `X`.

Tabla 6. Diseño e implementación del método descodificar

Diseño	Implementación

descodificar(X:cola)→X":cola
 
 	
	Queue **descodificar**(Queue X2){
	Queue X1 = T1(X2);
	Queue X = T2(X1);
	return X;
	}

Analizando los mensajes `X″`(mensaje de entrada) y `X′`(mensaje de salida), nos damos cuenta que los caracteres que se encuentran en posiciones impares en el mensaje `X″` forman la primera mitad del mensaje `X′`. Mientras los caracteres que se encuentran en posiciones pares del mensaje `X″` forman la segunda mitad del mensaje `X′`, pero en orden inverso.

Por ello, la estrategia a seguir en la `T3` es mover directamente al mensaje de salida los caracteres que están en posiciones impares en el mensaje de entrada. Estos caracteres forman la primera mitad del mensaje de salida. Mientras, los caracteres que están en posiciones pares los apilaremos en una pila, con el objetivo de darle la vuelta al orden en que aparecen. Finalmente, después de terminar de analizar todo el mensaje de entrada pasaremos todos los caracteres de la pila a la segunda mitad del mensaje de salida.

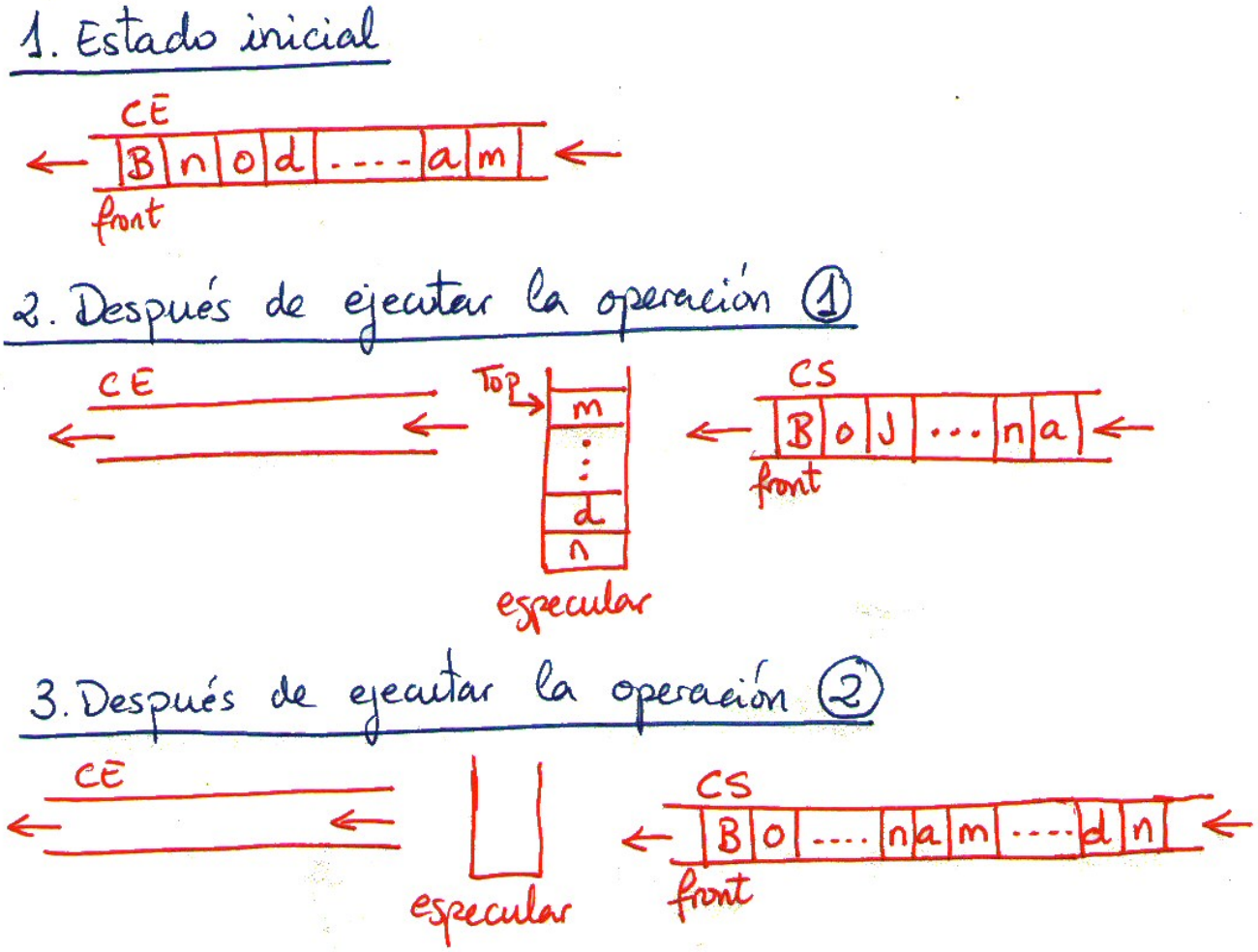
Tabla 7. Diseño e implementación del método T3

Algoritmo	Implementación

T3(CE:cola) → CS:cola
1. mientras haya caracteres en CE
a. car = extraer carácter de CE
b. si se encuentra en posición impar
encolar car e CS
si no
apilar car en una pila
fin si
fin mientras
2. mover todos los caracteres de la pila a CS
devolver CS
	Queue **T3**(Queue ce){
	Queue cs = new Queue();
	Stack especular = new Stack();
	while (!ce.isEmpty()){
	//posiciones impar
	cs.enqueue(ce.dequeue());
	//posiciones par
	if (!ce.isEmpty())
	especular.push(ce.dequeue());
	}
	moverDePilaACola(especular, cs);
	return cs;
	}

La Figura 3 describe los estados en los que se pueden encontrar las estructuras de datos utilizadas para la ejecución del método T3.

Figura 3. Estados de las estructuras de datos utilizadas en la ejecución del método T3



b)

La complejidad del método `codificar` será la suma de la complejidad de `T1` y `T2`, siendo el valor de ***n*** (que vamos a utilizar más adelante) el número de caracteres en el mensaje de entrada. Analizando la complejidad de `T1` nos damos cuenta que cada operación interna del bucle principal hay que analizarla individualmente en cada caso correspondiente, ya que el procesamiento o consumo de cada carácter del mensaje se realiza en los bucles internos (operaciones 1 y 2). Primeramente, vamos a detallar los diferentes casos, los cuales pueden ser el caso mejor o caso peor para cada operación.

El primer caso es cuando todos los caracteres del mensaje son vocales. El segundo caso es que ninguno de los caracteres del mensaje es vocal. El tercer caso sería cuando la mitad de los caracteres son vocales y la otra mitad no y además están colocados de forma intercalada, es decir, vocal, no vocal, vocal, no vocal y así sucesivamente. El último caso o caso general lo compone cualquier otro ejemplo.

Antes que nada comentar que las operaciones principales de las colas y pilas son de orden constante, es decir, O(1). Lo mismo que el método `esVocal()`. A continuación vamos a analizar la complejidad de cada operación de forma independiente para cada caso correspondiente:

Primer caso

1) *O(n) – **n+1** comparaciones y consultas, más los caracteres se extraen y encolan **n veces***
2) *O(1) – solo se ejecuta una vez*
3) *O(1) – solo se realiza la comparación*
4) *O(1) – la comparación del bucle principal sólo se ejecuta 2 veces*

Segundo caso

1) *O(1) – solo se ejecuta una vez*
2) *O(n) – **n+1** comparaciones y consultas, más los caracteres se extraen y apilan **n veces***
3) *O(n) – 1 comparación, **n+1** comparaciones, más los caracteres se desapilan y encolan **n veces***
4) *O(1) – la comparación del bucle principal sólo se ejecuta 2 veces*

Tercer caso

1) *O(n) – **n** comparaciones y consultas, más los caracteres se extraen y encolan **n/2 veces***
2) *O(n) –**n** comparaciones y consultas, más los caracteres se extraen y apilan **n/2 veces***
3) *O(n)–**n/2** comparaciones, **n** comparaciones, más los caracteres se desapilan y encolan **n/2 veces***
4) *O(n) – la comparación del bucle principal se ejecuta **n/2+1 veces***

Caso general

Analizando los 3 ejemplos extremos anteriores, nos damos cuenta que en el caso general las operaciones 1 y 2 son complementarias, es decir, entre las dos operaciones se procesan todos los caracteres del mensaje de entrada en una pasada. Por tanto, la suma de la complejidad de las dos operaciones es de orden lineal. En cuanto a la operación 3, ésta depende de la operación 2 y por tanto tiene la misma complejidad, ya que procesa el mismo número de caracteres.

Finalmente, determinamos que la complejidad del método `T1` es de orden lineal en todos los casos, dado que la suma total de la complejidad de sus operaciones es de orden lineal.

A continuación vamos a analizar la complejidad de `T2`. En este caso todos los ejemplos forman el caso general. El número de veces que se ejecutan las operaciones del bucle es de n/2 veces, dado que en cada paso se procesan dos caracteres (el primero y el último). Menos la operación 3a, las demás son de $O(1)$. Por lo tanto el coste total de estas operaciones es de $O(n)$, ya que se ejecutan n/2 veces.

Seguido pasamos a calcular la complejidad de la 3a (`extraerUltimoDeCola`). Esta operación es un poco especial, dado que el tamaño de la cola va decreciendo en cada llamada que se realiza. Por lo tanto, no podemos decir de forma directa que al tener un bucle (n-1 veces) sea de orden lineal. Por ello, las operaciones del método `extraerUltimoDeCola` las analizaremos en el contexto de la ejecución completa de `T2`. De esta forma vemos que las operaciones 1, 3 y 4, todas de $O(1)$, se ejecutan n/2 veces. Por lo tanto, su complejidad es $O(n)$. En cuanto a la operación 2, analizamos que la operación 2a se ejecuta al principio n-2 veces, luego n-4 veces y así hasta la última vez, en el que se ejecuta 2 veces. Por lo tanto, el coste de ejecutar 2a (cuya complejidad es de $O(1)$) lo da el siguiente sumatorio de n/2 términos: (n-2)+(n-4)+…+2. La suma total es n * n/4, lo que determina que su complejidad es de $O(n^2)$.

Finalmente, podemos concluir que la complejidad de `T2` es de $O(n^2)$ y lo determina la ejecución del método `extraerUltimoDeCola`. Por lo tanto, la complejidad del método `codificar` es de $O(n^2)$ y lo determina `T2`.

Del mismo modo que el método `codificar`, la complejidad del método `descodificar` es la suma de la complejidad de los métodos `T1` y `T3`. Nos falta analizar la complejidad de `T3`. El método `T3` se compone de dos operaciones principales. El primero de ellos procesa todos los caracteres de la cola en orden lineal, encolando la mitad en la cola de salida y la otra mitad en una pila. Por lo tanto $O(n)$. En cuanto a la segunda operación, procesa la mitad de caracteres del mensaje inicial, exactamente realiza n/2 operaciones de desapilar y encolar en la cola de salida. Por lo tanto su complejidad, también es de orden lineal. Después de analizar las dos operaciones principales, determinamos que `T3` es de $O(n)$. Finalmente, determinamos que el método `descodificar` es de $O(n)$, en base a que `T1` y `T3` ambos son de $O(n)$.

¿Hay alguna parte del proceso que se podría mejorar?

Vemos que el método `T2` es el proceso que merecía la pena volver a estudiarlo, para ver si se podría mejorar de $O(n^2)$ a $O(n)$.

En caso afirmativo, ¿Cómo lo mejorarías? Detalla cuales son las características (algoritmo más eficiente, nueva estructura de datos, operaciones más relevantes y su complejidad,…) de tu solución sin implementarla y calcula la nueva complejidad.

Se nos ocurren dos soluciones: 1) modificando el algoritmo y realizando una operación parecida a `T3` y 2) utilizando otra estructura de datos que no tenga las restricciones de la cola (p.ej: una lista enlazada doble).

El algoritmo o estrategia de la primera solución sería: 1) extraer y encolar la primera mitad de los caracteres en CE; 2) extraer la segunda mitad de CE y apilarlo en una pila para darle la vuelta al orden de los caracteres y 3) ir encolando en CS los caracteres de CE y la pila de forma alternativa, empezando como primer carácter el de la cola de CE, seguido de uno de la pila, y así sucesivamente. En la siguiente tabla se detalla más a fondo el algoritmo y la implementación:

Algoritmo	Implementación

T2(CE:cola) → CS:cola
1. repetir tamaño de CE/2 veces
extraer carácter de CE y encolarlo en CE
fin repetir
2. repetir tamaño de CE/2 veces
extraer carácter de CE y apilarlo en una pila
fin repetir
3. mientras haya caracteres en CE
extraer carácter de CE y encolarlo en CS
desapilar carácter de la pila y encolarlo en CS
fin mientras
devolver CS
	Queue **T2**(Queue ce){
	Queue cs = new Queue();
	Stack especular = new Stack();
	int n = ce.size();
	int n1 = (n mod 2==0)?n/2:(n/2)+1;
	int n2 = n/2;
	for(int i=0;i<n1;i++)
	ce.enqueue(ce.dequeue());
	for(int i=0;i<n2;i++)
	especular.push(ce.dequeue());
	while (!ce.isEmpty()){
	cs.enqueue(ce.dequeue());
	cs.enqueue(especular.pop());
	}
	return cs;
	}

La complejidad del método anterior es de orden lineal, ya que el coste de ejecución sería n/2 * (2 + 2 + 4)=**4n** (contando como coste 1 la ejecución de una operación principal de la pila o la cola). Comparándolo con la solución del apartado a) se baja el nivel de complejidad. Aquellos que hayan utilizado esta estrategia en el apartado a), difícilmente van a mejorar la complejidad de O(n) cambiando de algoritmo. La otra opción podría ser optar por cambiar de estructura de datos para la representación del mensaje. Esta es la segunda solución que se comenta en el siguiente párrafo.

En cuanto a la segunda solución, el problema principal es el método `extraerUltimoDeCola`. Para este problema nos interesaría que la complejidad de esta operación también fuera constante. Esto se podría conseguir extendiendo la funcionalidad de la cola en una nueva estructura de datos que la podríamos llamar (p.ej. **bicola**) por la característica de que los elementos que pertenecen a la **bicola** se podrían consumir de los dos extremos con un coste de ejecución constante. Una implementación óptima para esta estructura de datos sería utilizar internamente una lista doblemente enlazada. De está forma podríamos utilizar el siguiente algoritmo:

Algoritmo	
T2(CE:bicola) → CS:bicola mientras haya caracteres en CE car=extraer carácter del lado izquierdo de CE encolar car en CS por el lado derecho car=extraer carácter del lado derecho de CE encolar car en CS por el lado derecho fin mientras devolver CS	

La complejidad del método anterior también sería de orden lineal, dado que su coste de ejecución sería n/2 * 4 = **2n** (contando como coste 1 la ejecución de una operación principal de la bicola). Comparando con la primera solución vemos que el coste de ejecución baja a la mitad, pero utilizando la notación asintótica diríamos que los dos son de orden lineal, $O(n)$. Por lo tanto las dos soluciones mejoran la solución del apartado a), ya que se baja el nivel de complejidad de $O(n^2)$ a $O(n)$.