TEMA 1

Tipos Abstractos de Datos.

CONSIDERACIONES GENERALES.

- El TAD es oculto por naturaleza. Se desconoce cualquier referencia a su funcionamiento interno.
- Salvo expresa indicación en contra, las estructuras de datos no deberán ser modificadas como consecuencia de la ejecución de los subprogramas.
- Con carácter general, no se permitirá el empleo de estructuras de datos auxiliares y tan solo se admitirá un recorrido por las estructuras (p. ej. el conjunto apilar / desapilar).
- Por razones de eficiencia, tanto para el empleo de la técnica iterativa como para el de la recursiva se deberá prestar atención especial a la finalización anticipada, es decir no recorrer toda la estructura si no es necesario. Esto se consigue:
    - En técnicas iterativas declarando una condición de terminación compuesta.
    - En técnicas recursivas *no realizando posteriores llamadas*.
- Prestar especial atención a la cabecera del módulo. Habitualmente se penalizan los **siguientes errores:**
    - No incluir algún argumento imprescindible.
    - Utilizar más argumentos de los estrictamente necesarios, bien porque no vayan a utilizarse, o bien por ser ya accesibles (caso de variables globales o módulos internos).
    - No proteger la información utilizando variables globales**.**

- Otro aspecto importante a valorar es el relativo a la legibilidad (y en consecuencia a su mantenibilidad) del código. Teniéndolo en cuenta, se atenderá a consideraciones tales como:
    - Su brevedad.
    - La inexistencia de lógica redundante.
    - El empleo de nombres de variables significativos.
- Y, por supuesto, el funcionamiento correcto. El algoritmo deberá hacer lo que se pide y no otra cosa y tratar de contemplar todas las posibilidades, en particular las situaciones de excepción (por ejemplo, recibir la estructura vacía).

EJERCICIOS

PilasRepetirN

Enunciado.

Se dispone del TAD *Pila de Enteros*, que tiene implementadas las siguientes operaciones:

```
boolean pilaVacia ()
void apilar (int x)
int desapilar ()
```

SE PIDE:

Codificar un método en java que, recibiendo como parámetros una pila del tipo Pila y un número entero *n*, devuelva la misma pila, en la que cada elemento de la pila original se repita *n* veces.

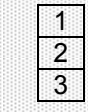


Figura 1

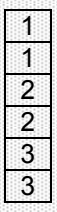


Figura 2

NOTA:

- Cada entero de la pila sólo podrá ser sacado una única vez.
- No se permite la utilización de estructuras de datos auxiliares.

Ejemplo:

Dada la pila de la figura 1 y *n*=2, después de ejecutar el método debe quedar como en la figura 2.

Orientación.

El tratamiento es recursivo. Se trata de vaciar la *pila* y en la fase de "vuelta" apilar el elemento tantas veces como indique el argumento *n*.

Código.

```
static void repetirN (Pila p, int n)  {
   int elem, i;
   if (! p.pilaVacia ()) {
      elem = p.desapilar ();
      repetirN (p, n);
      for (i = 0; i < n; i++)
         p.apilar (elem);
   }
}
```

PilasSumergirMenor

Enunciado.

Dado el TAD Pila de Números Enteros con las siguientes operaciones:

boolean pilaVacia ()

void apilar (int x)

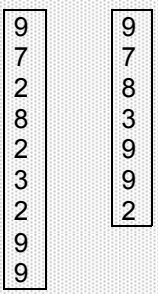
void desapilar ()

SE PIDE codificar un método en Java que, recibiendo como parámetro una pila perteneciente al TAD anterior, inserte en el fondo de dicha pila el elemento más pequeño que en ella esté contenido, eliminando además todos aquellos elementos que coincidan con dicho elemento más pequeño.

OBSERVACIONES:

- No se permite la utilización de estructuras de datos auxiliares.
- Cada elemento de la pila solamente podrá ser desapilado y apilado una vez.
- En caso de que la pila se encuentre vacía, la ejecución del método no deberá surtir ningún efecto.
- Además de la pila, la cabecera del método podrá incluir otros parámetros.

EJEMPLO: dada la pila mostrada en la parte izquierda de la figura, el método deberá dejar dicha pila en la situación mostrada en la parte derecha de la mencionada figura. Como puede apreciarse, el 2, que es el elemento más pequeño, ha sido insertado en el fondo de la pila, habiéndose eliminado además todas las repeticiones del 2.



Orientación.

Se realiza un tratamiento recursivo en cuya fase de "ida" se comprueba cuál es el elemento menor. Dicho elemento se pasa a la siguiente llamada recursiva en el argumento *menor*, (que se va a devolver como resultado del método, para poderlo consultar a la "vuelta").

Para inicializar dicho argumento es necesario que el tratamiento recursivo contemple como caso particular el primer elemento de la *pila* inicial. Para ello se deberá pasar un argumento adicional (*primero*), de tipo booleano que se inicializa a *true* desde un módulode llamada (*sumergirMenor),* no recursivo. Una vez ejecutado el tratamiento correspondiente a la primera llamada dicho argumento tomará de forma definitiva el valor *false* y el valor inicial de *menor* será el primer elemento de la *pila*.

En las llamadas siguientes a la primera se actualizará el valor del argumento *menor* en función del elemento (*elem*) desapilado.

En la fase de transición (cuando la *pila* esté vacía), apilamos en el fondo de la pila *menor, **pero no en el caso de recibir la pila inicialmente vacía (***esta circunstancia se detecta cuando el argumento *primero* es *true*), y lo devolvemos como resultado.

En la fase de vuelta se comparan los elementos de la *pila* con el resultado del método (que guardamos en la variable *resul*) y sólo apilamos los que sean diferentes de *resul*. Dado que debe conservarse el valor de *menor* con que se alcanzó la fase de transición debería pasarse por referencia, pero ello no es posible en Java, por lo que lo devolvemos como resultado del método.

La condición de finalización del tratamiento recursivo es la exploración completa de la *pila* (*pila.pilaVacia () == true*) y no existe posibilidad de terminación anticipada.

Código.

```
static int buscarMenor (Pila pila, int menor, boolean primero) {
    int elem, resul = 999;
    if (!pila.pilaVacia ()) {
        elem = pila.desapilar ();
        if (primero)
            menor = elem;
            else if (elem < menor)
                    menor = elem;
        resul = buscarMenor (pila, menor, false);
        if (elem != resul)
             pila.apilar (elem);
    }
    else if (!primero) {
             pila.apilar (menor);
             resul = menor;
         }
    return resul;
}
public static void sumergirMenor (Pila pila) {
         buscarMenor (pila, 999, true);
}
```

PilasInsertarCero

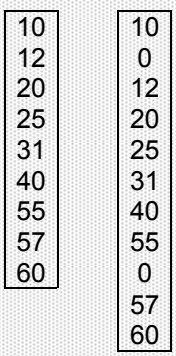
Enunciado.

Dado el TAD Pila de Números Enteros Positivos con las siguientes operaciones:

```
boolean pilaVacia ()
void apilar (int x)
int desapilar ()
```

SE PIDE:

Codificar un método en Java que, recibiendo como parámetro una pila perteneciente al TAD anterior y ordenada desde la cima hasta el fondo, inserte un cero entre aquellos elementos consecutivos de la pila cuya diferencia sea la mínima existente dentro de dicha pila.



OBSERVACIONES:

- No se permite la utilización de estructuras de datos auxiliares.
- Cada elemento de la pila solamente podrá ser desapilado y apilado una vez.
- Se supone que en la pila pasada como parámetro, no existen elementos repetidos.
- Sí la pila está vacía o posee un solo elemento, la ejecución del método no deberá surtir ningún efecto.

EJEMPLO:

Dada la pila mostrada en la parte izquierda de la figura, como puede apreciarse la mínima diferencia entre elementos consecutivos es 2 (12-10 y 57-55). En consecuencia, el método deberá dejar dicha pila en la situación mostrada en la parte derecha de la mencionada figura.

Orientación.

El núcleo del proceso es un tratamiento necesariamente recursivo (para poder cumplir las condiciones de recorrido único y no utilizar estructuras de datos auxiliares).

Dicho tratamiento necesitará conocer el valor del elemento desapilado en la llamada previa (*ant*,) para poder calcular la diferencia actual (el elemento desapilado (*elem*) actúa como minuendo y el recibido de la llamada anterior (*ant)* como sustraendo).

Así pues, el proceso de ida permite conocer el valor de la menor diferencia. El resultado de la menor diferencia (*minDif*) se debería pasar por referencia para su consideración en el proceso "de vuelta". Dado que no es posible en Java se realizará un método entero que devuelva como resultado el valor de *minDif* (que se pasasrá como argumento).

En caso de que varias parejas de elementos proporcionen igual diferencia mínima, el enunciado dice que hay que insertar cero entre todos ellos.

Con carácter excepcional deben tratarse:

- El primer elemento (si existe), dado que no hay anterior (será el primer *ant)*.
- El segundo elemento (si existe) a partir del cual se "inicializa" la primera diferencia mínima (*minDif*).

En consecuencia, se necesita un argumento complementario (*pos*) entero, que contenga la información del número de orden del elemento de la pila.

El procedimiento recursivo tendrá básicamente el siguiente esquema:

- En la fase de ida: se desapila, se actualiza el valor de *difMin* (se tratan excepcionalmente los elementos correspondientes a *pos*=1 y *pos*=2), y se realiza la llamada recursiva pasando como argumento *ant* el elemento (*elem*) actualmente desapilado e incrementando en una unidad el argumento *pos*.

- En la fase de transición se devuelve el resultado del método (el valor de *minDif* guardado en la variable *resul)*.
- En la fase de vuelta: se apila el elemento (*elem*) en cualquier caso, se verifica el cumplimiento de diferencia mínima (si *resul* == *elem* - *ant*) para, en su caso, insertar un 0 (si *pos*> **1)**

El módulo de llamada al procedimiento recursivo deberá inicializar (a 1) el argumento *pos* en tanto que es indiferente el valor con el que envíe al argumento *ant*.

Obsérvese que el algoritmo contempla las situaciaciones excepcionales de recibir la *pila* vacía o con un único elemento (en ambos casos deberá quedar inalterada). Así mismo, cuando la pila contenga sólo dos elementos deberá insertar un 0 entre ellos.

Código.

```
static int inserta0 (Pila pila, int ant, int pos, int minDif) {
    int elem, resul;
    if (!pila.pilaVacia ()) {
        elem = pila.desapilar ();
        if (pos > 2) {
            if ( (elem-ant) < minDif)
                minDif = elem-ant;
        }
        else if (pos == 2)
            minDif = elem-ant;
        resul = inserta0 (pila, elem, pos+1, minDif);
        pila.apilar (elem);
        if (pos > 1)
            if ( (elem-ant) == resul)
                pila.apilar (0);
        }
        else resul = minDif;
        return resul;
    }

    static void insertarCero (Pila pila) {
        inserta0 (pila, 77, 1, 77);
    }
```

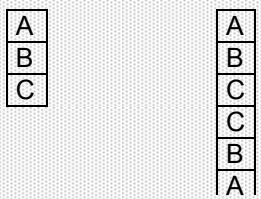
PilasHacerCapicua

Enunciado.

Se dispone del **TAD Pila** de caracteres con las siguientes operaciones:

```
void inicializarPila ()
boolean pilaVacia ()
void apilar (char x)
char desapilar ()
```

Dada una pila del TAD indicado anteriormente.



SE PIDE: Implementar un algoritmo que haga que una pila, que se recibirá como parámetro, se convierta en capicúa. Es decir, el primer elemento será igual que el último, el segundo igual que el penúltimo, etc.

NOTA: No se permitirá la utilización de ninguna estructura de datos auxiliar.

Orientación.

Se trata de desarrollar un método recursivo que permita, en la fase de ida vaciar la *pila*. A la vuelta habrá que colocar el elemento dos veces en la pila: primero se sumerge el dato, y luego se apila para conseguir la simetría pedida en el ejercicio.

Dado que no existe la operación *sumergir* será necesario construirla[1]. Esto implica el desarrollo de un tratamiento recursivo que, recibiendo como argumentos una *pila* y un *dato,* proceda como sigue:

- En la fase de ida se desapila hasta vaciar la estructura.
- En la transición (*pila.pilaVacia ()=true*) se apila el *dato*.
- A la vuelta se recuperan los elementos originales de la pila.

Código.

```
static void sumergir (Pila pila, int dato)  {
   int elem;
   if (!pila.pilaVacia ())  {
      elem = pila.desapilar ();
      sumergir (pila, dato);
      pila.apilar (elem);
    }
   else pila.apilar (dato);
}
static void hacerCapicua (Pila pila)  {
   int elem;
   if (!pila.pilaVacia ()) {
      elem = pila.desapilar ();
      hacerCapicua (pila);
      sumergir (pila, elem);
      pila.apilar (elem);
    }
}
```

[1] Esto implica necesariamente la necesidad explícita de apilar y desapilar más de una vez. (El enunciado no establece esta limitación)

PilasApilarReorganizable

Enunciado.

Dado el TAD Pila de números enteros con las siguientes operaciones:

```
void inicializarPila ()
boolean pilaVacia ()
void apilar (int x)
int desapilar ()
```

Se define PILA REORGANIZABLE como aquella en la que, a la hora de insertar un número n,

- Si n ya se encuentra en la pila, n es desplazado a la cima, conservando el resto de elementos su orden relativo.
- Si n no se encuentra en la pila, n es apilado en la cima.

SE PIDE implementar un método en Java que, recibiendo como parámetros una pila reorganizable perteneciente al TAD anterior y un número entero n, inserte dicho número n en la pila.

OBSERVACIONES:

- No se permite la utilización de estructuras de datos auxiliares.
- Cada elemento de la pila solamente puede ser desapilado y apilado una vez.
- Como es obvio, en la pila no deberán aparecer números repetidos.

Orientación.

El núcleo del proceso se basa en un método recursivo que recibe como argumentos la *pila* y el *dato*: se utiliza un método que llama a otro auxiliar recursivo, que (si existe) borra el *dato* de la *pila*.

El proceso admite terminación anticipada (no realizar nuevas llamadas recursivas) antes de alcanzar la condición pesimista (*pila.pilaVacia=true*) cuando se desapile algún elemento de valor == *dato*.

A la vuelta habrá que **apilar sólo** en caso de que el elemento sea distinto de *dato* y **no apilar** en caso contrario.

La etapa final consiste en apilar sobre la cima el *dato* desdel método de llamada.

Código.

```
static void eliminarDato (Pila pila, int dato)  {
   int elem;
   if (!pila.pilaVacia ()) {
      elem = pila.desapilar ();
      if (elem != dato) {
         eliminarDato (pila,dato);
         pila.apilar (elem);
      }
   }
}
static void apilarReorganizable (Pila pila, int dato)  {
   eliminarDato (pila,dato);
   pila.apilar (dato);
}
```

PilasBorrarEnP1ElementosDeP2

Enunciado.

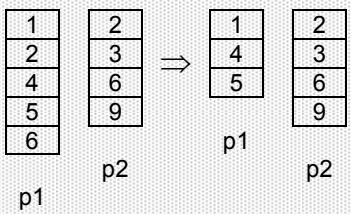
Dado el TAD Pila de Números Enteros con las siguientes operaciones:

```
boolean pilaVacia ()
void apilar (int x)
int desapilar () ;
```

SE PIDE codificar un método en Java que, recibiendo como parámetros dos pilas, p1 y p2, pertenecientes al TAD anterior y ordenadas ascendentemente desde la cima hasta el fondo, borre de la pila p1 todos aquellos elementos que se encuentren en p2

OBSERVACIONES:

- No se permite la utilización de ninguna estructura de datos auxiliar.
- Cada elemento perteneciente a cada una de las dos pilas, sólo podrá ser desapilado y apilado una vez.
- Al finalizar la ejecución del método, la pila p2 deberá quedar en el mismo estado inicial, es decir con los mismos elementos y en el mismo orden ascendente.
- Asimismo, al finalizar la ejecución del método, la pila p1 deberá quedar sin los elementos contenidos en p2, pero conservando también el orden ascendente inicial.



EJEMPLO: Dadas las pilas p1 y p2 mostradas en la parte izquierda de la figura, al finalizar la ejecución del método dichas pilas deberán quedar en la situación mostrada en la parte derecha de la mencionada figura.

Orientación.

El ejercicio es una variante del algoritmo básico de enfrentamiento de *pilas*, la diferencia principal reside en el hecho de que, en este caso, la finalización del proceso tiene lugar cuando se alcanza el final de cualquiera de ellas (solo se ejecuta si (*aux1 && aux2)* == *true*, sin posibilidad de terminación anticipada) y no se genera una tercera *pila*.

En esencia, el algoritmo consiste en avanzar (desapilando) por *pila1* o *pila2* en función del resultado de comparar los elementos desapilados (*elem1* y *elem2*). En la fase de transición ((*pend1 && pend2*) == *false*), si hay algún elemento pendiente de apilar, se apila en su *pila* correspondiente. En la fase de vuelta **no se apilará** en *pila1* (lo que equivale a eliminar) en caso de coincidencia de *elem1* y *elem2*.

Los argumentos son los ya conocidos *pila1* y *pila2*, los enteros *elem1* y *elem2* y las variables booleanas *apilar1* y *apilar2*.

Código.

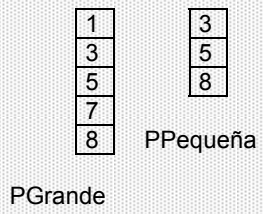
```
static void borrarEnP1 (Pila pila1, Pila pila2, boolean apilar1, boolean apilar2, int elem1, int elem2) {
   boolean pend1 = !pila1.pilaVacia () || apilar1;
   boolean pend2 = !pila2.pilaVacia () || apilar2;
   if (pend1 && pend2) {
      if (!apilar1)
         elem1 = pila1.desapilar ();
      if (!apilar2)
         elem2 = pila2.desapilar ();
      if (elem1 < elem2) {
         borrarEnP1 (pila1, pila2, false, true, elem1, elem2);
         pila1.apilar (elem1);
       }
      else if (elem2 < elem1) {
            borrarEnP1 (pila1, pila2, true, false, elem1, elem2);
            pila2.apilar (elem2);
         }
         else {
            borrarEnP1 (pila1, pila2, false, false, elem1, elem2);
            pila2.apilar (elem2);
         }
   }
   else if (apilar1)
            pila1.apilar (elem1);
         else if (apilar2)
               pila2.apilar (elem2);
}
static void borrarEnP1ElementosDeP2 (Pila pila1, Pila pila2)  {
   borrarEnP1 (pila1, pila2, false, false, 0, 0);
}
```

PilasEstaContenida

Enunciado.

Dado el *TAD Pila* con las siguientes operaciones:

```
boolean pilaVacia ()
void apilar (int x)
int desapilar () ;
```



SE PIDE codificar un método booleano en Java que, recibiendo como parámetros dos pilas, PPequeña y PGrande, pertenecientes al TAD anterior, y ordenadas ascendentemente desde la cima hacia el fondo, determine si todos los elementos de PPequeña se encuentran contenidos en PGrande.

OBSERVACIONES:

- No se permite la utilización de ninguna estructura de datos auxiliar.
- Cada uno de los elementos pertenecientes a las dos pilas solamente podrá ser apilado y desapilado una vez.
- Se supone que ninguna de las dos pilas está vacía.
- Se supone que, dentro de una misma pila, no existen claves repetidas.

EJEMPLO: Dadas las pilas mostradas en la figura, el método deberá devolver *true*, ya que todos los elementos de PPequeña se encuentran contenidos en PGrande.

Orientación.

En este problema tenemos que comprobar si todos los elementos de la pila pequeña están dentro de la pila grande. Para verificarlo, utilizaremos como argumentos del método las dos pilas (*pGrande y pPeque*), el último valor extraido de la pila pequeña (*p*), y una variable booleana que indica si necesitamos sacar un elemento de la pila pequeña (*sacar*, pasado por valor):

- Si existen elementos por tratar en la pila pequeña (la pila está vacía, o bien *sacar=false* y por tanto tenemos *p* pendiente de tratar):
  - Si *sacar=true*, desapilamos de *pPeque*.
  - Si *pGrande* no está vacía, sacamos un elemento (*g*),y a continuación:
    - Si el elemento de la pila grande *(g)* es menor que el de la pila pequeña *(p)* se produce una llamada recursiva pasando el elemento *p*, e indicando que no tenemos que sacar ningún elemento (*sacar=false*)
    - Si ambos elementos son iguales (*g = p)* se produce una llamada recursiva solicitando despilar de *pPeque* (*sacar = true*).
    - Si *g>p* se produce la **terminación anticipada** y el resultado del método será *false*.
    - En la fase de "vuelta" habrá que apilar siempre en *pGrande*.
  - Si *pGrande* está vacía, al menos un elemento de la pila pequeña no está en la pila grande, y por tanto, devolveremos como resultado de del método *false*.
  - Si *sacar=true*, a la vuelta apilamos en ppeque el elemento *p*.
- Si no quedan elementos por tratar en *pPeque*, no importa el contenido de *pGrande*, y el resultado devuelto por el método será *true*.

Código.

```
static boolean contenida (Pila pPeque, Pila pGrande, int p, boolean sacar) {
   int g;
   boolean aux;
   if (!pPeque.pilaVacia () || !sacar) {
      if (sacar)
         p = pPeque.desapilar ();
      if (!pGrande.pilaVacia ()) {
         g = pGrande.desapilar ();
         if (p == g)
            aux = contenida (pPeque, pGrande, p, true);
         else if (p < g)
               aux = false;
               else aux = contenida (pPeque, pGrande, p, false);
         pGrande.apilar (g);
       }
      else aux = false;
      if (sacar)
         pPeque.apilar (p);
    }
   else aux = true;
   return aux;
}
static boolean estaContenida (Pila pilaPeque, Pila pilaGrande)  {
   return contenida (pilaPeque, pilaGrande, 77, true);
}
```

PilasVectorPilasMezclaOr

Enunciado.

Dado el Tad Pila de Números Enteros con las siguientes operaciones:

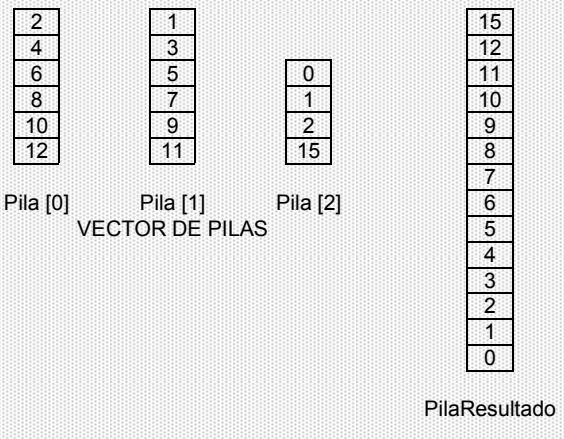
boolean pilaVacia ()
/*Devuelve *true* si la Pila pasada como parámetro está vacía y *false* en caso contrarío*/
void apilar (int x)
/*Introduce en la Pila un nuevo Elemento, convirtiéndose éste en la nueva cima de la Pila*/
int cima ()
/*Devuelve el último elemento apilado en la Pila, es decir, el elemento que constituye la cima de la Pila, pero sin desapilarlo*/
void decapitar ()
/*Elimina de la Pila el último elemento introducido, es decir, el que constituye la cima de la pila*/
void inicializarPila ()
/*Crea una Pila que inicialmente estará vacía*/

SE PIDE codificar un método en Java que, recibiendo como parámetro un vector de N pilas, ordenadas ascendentemente desde la cima hasta el fondo, perteneciente al tipo anterior, construya otra pila, ordenada descendentemente desde la cima hasta el fondo, que constituya la mezcla ordenada de los elementos contenidos en el vector de pilas.

OBSERVACIONES:

- No se permite la utilización de ninguna estructura de datos auxiliar.
- Cada elemento contenido tanto en cada una de las pilas del vector, como en la pila resultado, solamente podrá ser desapilado y apilado una vez.
- Se supone que en ninguna de las pilas del vector existen elementos repetidos.
- Asimismo, en la pila resultado tampoco deberán existir elementos repetidos.

EJEMPLO: Dado N = 3 y el vector de pilas mostrado en la figura, el método deberá generar la pila resultado indicada



Orientación.

Se trata de explorar el conjunto de cimas del vector de pilas y seleccionar la menor**.** Este elemento será el que haya que insertar (apilar) en la pila resultante (*pila*). La condición de terminación (no existe terminación anticipada) es que se haya realizado la exploración completa de todas las pilas del vector.

La pila resultante (*pila*) no puede tener elementos repetidos. Para controlar esta circunstancia, antes de apilar cualquier elemento, verificaremos utilizando el método *cima* que dicho elemento es estrictamente mayor que el último elemento que hemos apilado.

Para la resolución del ejercicio es necesario disponer de un método auxiliar (*posCimaMenor*) que, recibiendo como argumento el vector de pilas, devuelva un valor entero indicando el valor del índice del vector correpondiente a la pila cuya cima es menor (la primera si hay más de una).

Excepcionalmente, *pos* tomará el valor -1, que deberá interpretarse en el sentido de que todas las pilas están vacías y se utilizará como condición de terminación del tratamiento recursivo.

Código.

```
static int posCimaMenor (Pila [ ] vectorDePilas)  {
   int cimaMenor = 99999, pos, i;
   boolean primero = true;
   pos = -1;
   for (i = 0; i < N; i++) {
      if (!vectorDePilas[i].pilaVacia ())  {
         if (primero) {
            cimaMenor = vectorDePilas[i].cima ();
            pos = i;
            primero = false;
         }
         else if (vectorDePilas[i].cima () < cimaMenor) {
               cimaMenor = vectorDePilas[i].cima ();
               pos = i;
             }
      }
   }
   return pos;
}
static void vectorPilasMezclaOr (Pila [ ] vectorDePilas, Pila pila, int apilado, boolean primero)  {
   int elem, pos;
   pos = posCimaMenor (vectorDePilas);
   if (pos > -1) {
      elem = vectorDePilas [pos].cima ();
      vectorDePilas [pos].decapitar ();
      if (primero || (elem > apilado))
         pila.apilar (elem);
      vectorPilasMezclaOr (vectorDePilas, pila, elem, false);
      vectorDePilas [pos].apilar (elem);
   }
}
```

ColasContarElementos.

Enunciado.

Dado el TAD Cola de Números Enteros con las siguientes operaciones:

```
void inicializarCola ()
boolean colaVacia ()
void encolar (int x)
int desencolar ()
int primero ()
/*Devuelve el primer elemento de la Cola sin desencolarlo*/.
```

SE PIDE: Implementar un método entero en Java que, recibiendo como parámetro una cola perteneciente al tipo anterior, devuelva el número de elementos de la misma.

OBSERVACIONES:

- Los elementos de la cola no están ordenados.
- La cola no contiene elementos repetidos.
- Una vez finalizado el proceso, los elementos de la cola deberán quedar en el orden inicial.
- No se permite utilizar ninguna estructura de datos auxiliar.
- Sólo se permite la realización de un único recorrido en la cola.

Orientación.

Tratamiento iterativo.

La ejecución del proceso requiere un tratamiento a partir de ciclos *desencolar-encolar*, hasta recorrer la totalidad de los elementos de la *cola*, que deberán quedar en el orden inicial.

Inicialmente necesitamos conocer el primer elemento de la *cola* (variable *elemPr*) para actuar como "testigo" y finalizar el proceso cuando vuelva a aparecer (la *cola* no tiene elementos repetidos).

La solución deberá contemplar los casos singulares de recibir la *cola* vacía o con un solo elemento.

En la solución propuesta el tratamiento iterativo tiene lugar si la *cola* no está vacía (al menos tiene un elemento). Mediante un ciclo *while* se incrementa el contador de elementos siempre que el primer elemento de la *cola* sea distinto de *elemPr* (se deberá usar la operación *primero*, pues si se emplea *desencolar* la *cola* habría quedado descolocada).

Existen otras posibles soluciones, una especialmente interesante sería realizar el control del ciclo mediante la sentencia *repeat*.

Código.

```
static int contarElementosIterativo (Cola cola)  {
   int res, elem, elemPr;
   if (!cola.colaVacia ()) {
      elemPr = cola.desencolar ();
      cola.encolar (elemPr);
      res = 1;
      while (cola.primero () != elemPr) {
         res++;
         elem = cola.desencolar ();
         cola.encolar (elem);
      }
   }
   else res = 0;
   return res;
}
```

Tratamiento recursivo.

El bloque recursivo (fase de ida) consiste en realizar la secuencia *desencolar*-*encolar*-llamar (incrementando en 1). La terminación (no existe posibilidad de que sea anticipada) tiene lugar cuando el primer elemento coincide con el primer elemento de la *cola* original (*elemPr)* que se pasará por valor. En ese momento (fase de transición) se inicializa el primer valor del método (0 ó 1, según cómo se haya planteado). No se realiza ningún tratamiento en la fase de vuelta.

En la solución propuesta se contempla un bloque que se ejecuta, si procede, una única vez para identificar el valor de *elemPr*. Esto requiere un argumento booleano, *prim*, inicializado a *true* desde el módulo de llamada, y cuyo valor cambia a *false* tras ejecutar la primera instancia del algoritmo. (Una solución alternativa consistiría en obtener el valor de *elemPr* desde un módulo de "lanzamiento" no recursivo. En tal caso, el argumento *prim* no sería necesario).

El método debe contemplar la circunstancia excepcional de que la *cola* se reciba vacía (y devolver 0 como resultado). Para ello se utiliza la operación *colaVacia* (que no es la condición de terminación del tratamiento recursivo).

Código.

```
static int contarElementosRecursivo (Cola cola, int elemPr, boolean prim)  {
   int res, elem;
   if (!cola.colaVacia ()) {
      if (prim) {
         elem = cola.desencolar ();
         cola.encolar (elem);
         res = 1 + contarElementosRecursivo (cola, elem, false);
      }
      else {
         if (elemPr != cola.primero ())    {
            elem = cola.desencolar ();
            cola.encolar (elem);
            res = 1 + contarElementosRecursivo (cola, elemPr, false);
         }
         else res = 0;
      }
   }
   else res = 0;
   return res;
}
```

ColasEliminarKesimo.

Enunciado.

Dado el TAD *Cola* con las siguientes operaciones:

```
boolean colaVacia ()
void encolar (int x)
int desencolar ()
```

SE PIDE:

Codificar un método en Java que, recibiendo una cola perteneciente al TAD anterior y un número entero k, suprima de dicha cola el elemento que ocupa el lugar k-ésimo. Ejemplo: Dada la cola de la figura (a) y el número 7, el método deberá devolver la cola mostrada en la figura (b).

Figura a

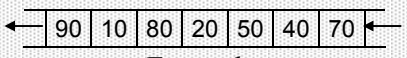


Figura b

OBSERVACIONES:

- No se permitirá el uso de ninguna estructura de datos auxiliar.
- Los restantes elementos de la cola deberán conservar su orden relativo inicial, tal y como se muestra en el ejemplo.

Orientación.

Dado que no se dispone de la operación *NumElemCola* no es posible resolver el ejercicio recorriendo la *cola* una sola vez (esquema *desencolar – encolar*), no obstante el enunciado no limita esta posibilidad. Se plantean dos alternativas:

- Desarrollar un método (*invertir*) que invierta el contenido de la *cola* y utilizar la técnica recursiva.
- Desarrollar un método (*numeroElementos*) que calcule el número de elementos de la *cola* y utilizar la técnica iterativa avanzando hasta alcanzar la posición que se desea eliminar

Orientación.

La condición de finalización "pesimista" será llegar al final (*cola.colaVacia () ==true*): se ha introducido un valor del argumento *pos* superior al número de elementos de la *cola*.

En condiciones normales se producirá una finalización anticipada al desencolar el elemento que ocupa la posición *pos* (se puede evitar un argumento disminuyendo *pos* en cada llamada recursiva) y no re-encolarlo. En este momento deberá invertir lo que quede de la *cola*. En la fase de vuelta se re-encolarán los elementos previamente desencolados.

Al finalizar el tratamiento recursivo se obtiene un resultado correcto pero con los elementos en orden inverso. Deberá ejecutar externamente al tratamiento recursivo el método *invertir*.

Los argumentos utilizados serán la *cola* y la *posición*.

Código.

```
static void invertir (Cola cola)  {
   int e;
   if (!cola.colaVacia ()) {
      e = cola.desencolar ();
      invertir (cola);
      cola.encolar (e);
   }
}
static void suprimir (Cola cola, int posicion)  {
   int elem;
   if (!cola.colaVacia ()) {
      posición--;
      elem = cola.desencolar ();
      if (posición > 0)
         suprimir (cola, posicion);
      else invertir (cola);
      if (posición != 0)
         cola.encolar (elem);
   }
}
static void eliminarKesimo (Cola cola, int k)  {
   suprimir (cola,k);
   invertir (cola);
}
```

ColasInsertarReorganizable.

Enunciado.

Dado el TAD cola de números enteros con las siguientes operaciones:

```
boolean colaVacia ();
void encolar (int x);
int desencolar () ;
void invertirCola ()
```

/*Operación que, recibiendo como parámetro una cola, invierte su contenido. La ejecución de esta operación no implica la realización de encolamientos y desencolamientos adicionales*/.

Se define Cola Reorganizable como aquella en la que, a la hora de insertar un elemento, se siguen los siguientes criterios:

- Si el elemento a insertar ya existe en la cola, dicho elemento es trasladado a la primera posición de la cola. Por ejemplo, dada la cola mostrada en la figura 1, si en ella se inserta un cero, la cola resultante sería la mostrada en la figura 2.
- Si el elemento a insertar no existe en la cola, dicho elemento es colocado en la primera posición de la cola. Por ejemplo, dada la cola mostrada en la figura 3, si en ella se inserta un cero, la cola resultante sería la mostrada en la figura 4.

SE PIDE: implementar un método en Java que, recibiendo como parámetros una cola reorganizable perteneciente al TAD anterior y un número entero x, inserte dicho número en la cola.

OBSERVACIONES:

- No se permite la utilización de ninguna estructura de datos auxiliar.
- Cada elemento de la cola solamente podrá ser encolado y desencolado una vez.

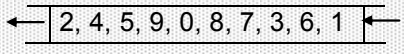


Figura 1

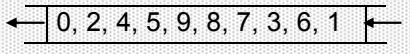


Figura 2

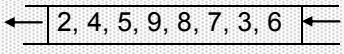


Figura 3

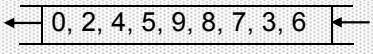


Figura 4

Orientación.

El método deberá recibir como argumento desde el módulo de llamada la *cola*. Dado que no pueden existir elementos repetidos (suponemos que todas las inserciones se han realizado siguiendo lo especificado en el enunciado), existe posibilidad de terminación anticipada. El método consistirá únicamente en recorrer la *cola* comprobando si el elemento a insertar existe. En caso de localizarlo, se aplicará la terminación anticipada, no volviéndolo a insertar, e invertiremos lo que queda de *cola*.

En el módulo de llamada (*insertarReorganizable*), se realizará la llamada al método *suprimirX* para eliminar el elemento *X*, se encolará dicho elemento, y luego se invertirá la *cola*.

Código.

```
static void suprimirX (Cola cola, int x)  {
    int elem;
    if (!cola.colaVacia ()) {
        elem = cola.desencolar ();
        if (elem != x) {
            suprimirX (cola, x);
            cola.encolar (elem);
        }
        else cola.invertirCola ();
    }
}
static void insertarReorganizable (Cola cola, int x)  {
    suprimirX (cola, x);
    cola.encolar (x);
    cola.invertirCola ();
}
```

ColasVerificarCremallera.

Enunciado.

Dado el TAD Cola de Números Enteros con las siguientes operaciones:

```
boolean colaVacia ();
void encolar (int x);
int desencolar () ;
int numElemCola ();
/*Método que, recibiendo como parámetro una cola, devuelve su número de elementos. La utilización de esta operación no implica la realización de encolamientos y desencolamientos adicionales*/
```

Se dice que dos colas ordenadas ascendentemente constituyen una cremallera si, y sólo si, al realizar una mezcla ordenada de ambas, se obtiene una secuencia de claves ascendente y consecutiva. Así, las dos colas mostradas en la figura *a)* constituyen una cremallera, ya que, como puede apreciarse, al realizar una mezcla ordenada de ambas, se obtiene la secuencia de claves 1, 2, 3, 4, 5, 6, 7, 8 y 9, que es una secuencia ascendente y consecutiva. Así mismo, las dos colas mostradas en la figura *b)* también constituyen una cremallera, ya que, al realizar una mezcla ordenada de ambas, se obtiene la secuencia de claves 2, 3, 4 y 5, que es una secuencia ascendente y consecutiva.

Sin embargo, las dos colas mostradas en la figura *c)* no constituyen una cremallera, ya que, al realizar una mezcla ordenada de ambas, se obtiene la secuencia de claves 0, 2, 3, 4, 5, 6, 7, 8 y 9, la cual no es consecutiva (obsérvese la no consecutividad entre el 0 y e1 2). Igualmente, las dos colas mostradas en la figura *d)* tampoco constituyen una cremallera, ya que, al realizar una mezcla ordenada de ambas, se obtiene la secuencia de claves 1, 2, 3, 4, 5, 6, 7, 7, 8 y 9, que no es una secuencia consecutiva, puesto que el 7 aparece repetido.

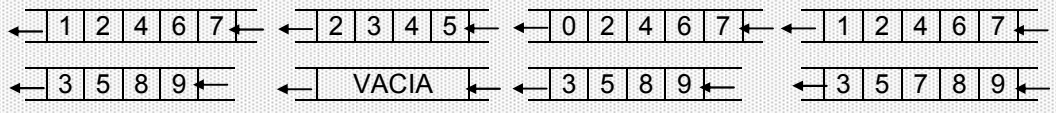


fig. a) fig. b) fig. c) fig. d)

SE PIDE: Implementar un método booleano en Java que, recibiendo como parámetros dos colas ordenadas ascendentemente, determine si constituyen una cremallera.

OBSERVACIONES:

- Cada elemento perteneciente a cada una de las colas, sólo podrá ser desencolado y encolado una vez.
- No se permite la utilización de estructuras de datos auxiliares.

Orientación.

La solución propuesta emplea la técnica iterativa dado que se dispone del método *numElemenCola*. Contempla tres bloques:

- Inicialización.
  - Se obtiene el número de elementos de cada una de las colas.
  - Se desencola de ambas colas para seleccionar el menor de ellos (variable *ant*), verificando la posibilidad de que ambos elementos sean iguales (en ese caso el método devuelve el valor *false*).
- Iteración principal. Tiene lugar mientras existan elementos por tratar en ambas colas.
  - Se verifica si el menor de los elementos a comparar es *ant*+1.
    - En caso afirmativo se continúa con la iteración re-encolando el elemento menor. Si se trata la totalidad de elementos de ambas colas el método devolverá el valor *true*.
    - En caso contrario el método devuelve el valor *false* y no se continúa con la iteración.
- Tratamiento final. Se procede a realizar el conjunto necesario de operaciones *desencolar – encolar* para dejar ambas estructuras en el orden inicial.

Código.

```
static boolean verificarCremallera (Cola cola1, Cola cola2) throws Colavacia {
    boolean  resul = true;
    int elem1 = 0, elem2 = 0, ant = 0,n1,n2;
    n1 = cola1.numElemCola ();
    n2 = cola2.numElemCola ();
    if (n1>0)
        elem1 = cola1.desencolar ();
    n1 = n1-1;
    if (n2>0)
        elem2 = cola2.desencolar ();
    n2 = n2-1;
    if ( (n1>=0) && (n2 < 0))
        ant = elem1-1;
    if ( (n2>=0) && (n1 < 0))
        ant = elem2-1;
    if ( (n1>=0) && (n2>=0))
        if (elem1<elem2)
            ant = elem1-1;
        else if (elem2<elem1)
                ant = elem2-1;
             else resul = false;
    while ( ( (n1>=0) && (n2>=0)) && resul)
        if (elem1<elem2) {
            cola1.encolar (elem1);
            if (elem1 == ant+1)
                ant = elem1;
            else resul = false;
            if (n1 > 0)
                elem1 = cola1.desencolar ();
            n1 = n1-1;
        }
        else if (elem1 > elem2) {
                cola2.encolar (elem2);
                if (elem2 == ant+1)
                    ant = elem2;
                else resul = false;
                if (n2 > 0)
                    elem2 = cola2.desencolar ();
                n2 = n2-1;
             }
             else {
                resul = false;
                cola1.encolar (elem1);
                if (n1 > 0)
                    elem1 = cola1.desencolar ();
                    n1 = n1-1;
                cola2.encolar (elem2);
                if (n2 > 0)
                    elem2 = cola2.desencolar ();
                    n2 = n2-1;
             }
    while (n1>=0) {
```

```
            cola1.encolar (elem1);
            if (elem1 != (ant+1))
                resul = false;
            ant = elem1;
            if (n1 > 0)
                elem1 = cola1.desencolar ();
            n1 = n1-1;
        }
        while (n2>=0) {
            cola2.encolar (elem2);
            if (elem2 != (ant+1) )
                resul = false;
            ant = elem2;
            if (n2 > 0)
                elem2 = cola2.desencolar ();
            n2 = n2-1;
        }
        return resul;
    }
```

ColasVectorColasMezclaOr.

Enunciado.

Dado el Tad Cola de Números Enteros con las siguientes operaciones:

```
void inicializarCola ();
boolean colaVacia ();
int primero () ;
void encolar (int x);
int desencolar () ;
void invertirCola () ;
```

Y la siguiente declaración de un vector de colas:

```
Cola [ ] vectorDeColas;
```

SE PIDE:

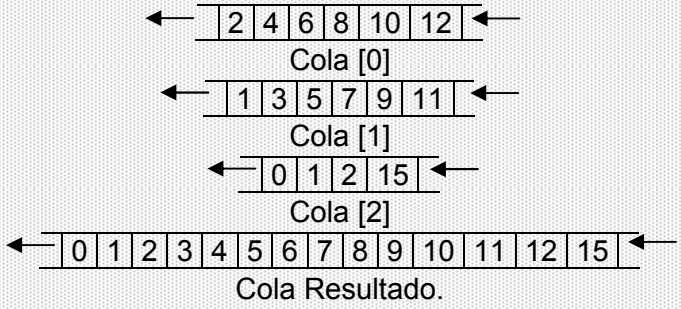
Codificar un método en Java que, recibiendo como parámetro un vector de colas, ordenadas ascendentemente desde el primer elemento (por orden de llegada) hasta el último, perteneciente al tipo anterior, construya otra cola, ordenada ascendentemente desde el primer elemento hasta el último, que constituya la mezcla ordenada de los elementos contenidos en el array de colas.

OBSERVACIONES:

- No se permite la utilización de ninguna estructura de datos auxiliar.
- Cada elemento contenido en cada una de las colas del vector, solamente podrá ser desencolado y encolado una vez. Así mismo, sólo se encolarán una vez los elementos de la cola resultado.
- Se supone que en ninguna de las colas del vector existen elementos repetidos. De igual forma, en la cola resultado no deberán aparecer elementos repetidos
- Al finalizar el método, las colas del vector deberán quedar en la misma situación inicial.

EJEMPLO:

Dado el vector de colas mostrado en la figura, el método deberá generar la cola resultado indicada.



Orientación.

Dado que la aplicación de un tratamiento recursivo invertirá el orden de las *colas* se necesita que el módulo de llamada (*vectorColasMezclaOr*) invierta todas ellas.

El método recursivo *mezclaOr* hace uso de la llamada al auxiliar *menor* (que no implica operaciones de *desencolar-encolar*) que devuelve el índice del vector que indica la *cola* que contiene el valor menor (el último si existen varios iguales). Excepcionalmente devuelve -1 cuando todas las *colas* se encuentran vacías.

La condición de terminación (no existe terminación anticipada) es que se encuentren vacías todas las *colas* del vector. Mientras tanto, se desencola el elemento correspondiente a la *cola* de índice proporcionado por el método auxiliar *menor*. En la fase de vuelta se encolará el elemento en la *cola* correspondiente, así como en la *cola* resultante siempre y cuando resulte mayor que el último encolado (para evitar repeticiones). Se obtiene así una *cola* resultante ordenada en sentido ascendente.

Código.

```
static int menor (Cola [ ] vectorColas)  {
   int i, aux = -1, min = Integer.MAX_VALUE, dato;
   for (i = 0; I < N; i++)
      if (! vectorColas [i].colaVacia ()) {
         dato = vectorColas [i].primero ();
         if (dato <= min) {
            min = dato;
            aux = i;
         }
      }
   return aux;
}
static void mezclaOr (Cola [] vectorDeColas, Cola colaR, boolean primer,int ant)  {
   int elem, aux;
   aux = menor (vectorDeColas);
   if (aux != -1) {
      elem = vectorDeColas [aux].desencolar ();
      if (primer || (elem != ant))
         colaR.encolar (elem);
      ant = elem;
      mezclaOr (vectorDeColas, colaR, false, ant);
      vectorDeColas [aux].encolar (elem);
   }
}
static void vectorMezclaOr (Cola [ ] vectorDeColas, Cola colaR)  {
   int ant = 0, i;
   colaR.inicializarCola ();
   mezclaOr (vectorDeColas, colaR, true, ant);
   for (i = 0; i< N; i++)
      vectorDeColas[i].invertirCola ();
}
```

PilasColasPalindromo.

Enunciado.

Dado el TAD Cola de caracteres con las siguientes operaciones:

```
boolean colaVacia ();
void encolar (int x);
int desencolar () ;
```

Y el TAD Pila de caracteres con las siguientes operaciones:

```
boolean pilaVacia ()
void apilar (int x)
int desapilar ()
```

SE PIDE:

Implementar un método booleano en Java que, recibiendo como parámetros una *cola* y una *pila* pertenecientes a los tipos anteriores e inicializadas ambas con un conjunto ***idéntico*** de caracteres, devuelva *true* si el citado conjunto de caracteres constituye un palíndromo[2]:

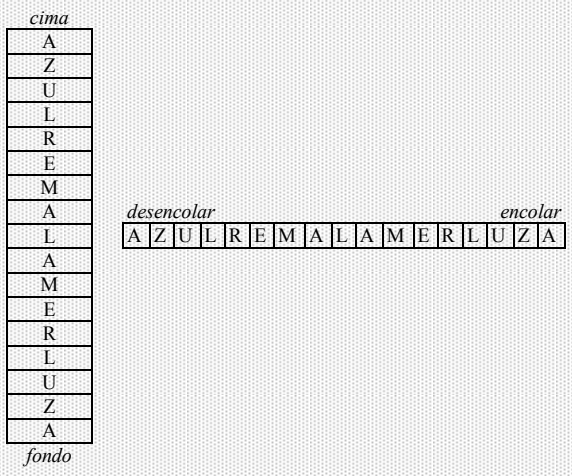
OBSERVACIONES:

- Al finalizar el proceso tanto la pila como la cola deberán permanecer con la estructura y contenidos iniciales.
- No se permite utilizar ninguna estructura de datos auxiliar.

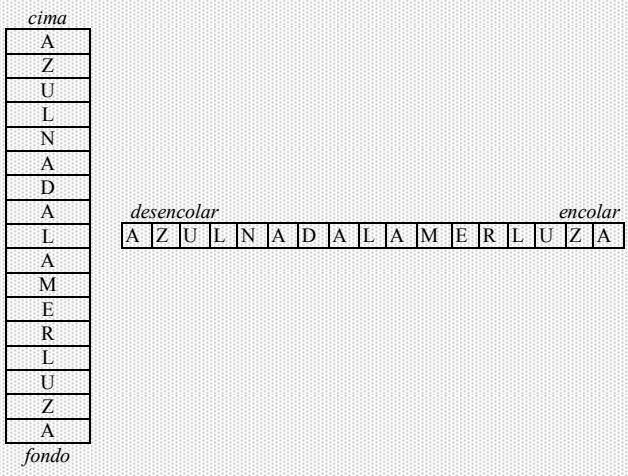
EJEMPLOS:

En el ejemplo *a)* el conjunto de caracteres constituye un palíndromo: el método devolverá *true*. No sucede así en el caso ejemplo *b*) en el que el método devolverá *false*

Ejemplo *a)*



Ejemplo *b)*



[2] **Palíndromo .** m. Palabra o frase que se lee igual de izquierda a derecha, que de derecha a izquierda; p. ej., *anilina; dábale arroz a la zorra el abad. (Diccionario RAE 22ª edición)*

Orientación.

Dado que **se sabe que el número de caracteres** de ambas estructuras **es idéntico**, el método recursivo propuesto, que no requiere más argumentos que sendas referencias a la *pila* y a la *cola*, consiste en:

- Fase de ida: se desapilar un elemento. La condición de terminación es *pila.pilaVacia () == true* y no es posible terminación anticipada.
- Fase de transición: Se inicializa (a *true*) el valor devuleto por el método.
- Fase de vuelta: (además de apilar) Se procesa cada elemento de la *cola* (secuencia *desencolar-encolar*) comparando ambos (*elemP* y *elemC*). En caso de que alguna pareja sea diferente se cambia (o reitera) el valor recibido a *false* y se devuelve a la instancia precedente.

Código.

```
static boolean esPalindromo (Pila pila, Cola cola) {
   boolean resul;
   char elemP, elemC;
   if (!pila.pilaVacia ()) {
      elemP = pila.desapilar ();
      resul = esPalindromo (pila, cola);
      pila.apilar (elemP);
      elemC = cola.desencolar ();
      cola.encolar (elemC);
      if (elemP != elemC)
         resul = false;
   }
   else resul = true;
   return resul;
}
```

OtrosTADsSimularVectorConCola.

Enunciado.

Dado el TAD Cola de caracteres con las operaciones que a continuación se indican:

```
boolean colaVacia ();
void encolar char (x);
char desencolar () ;
```

Se pretende simular el comportamiento de un Vector de N elementos con una estructura de tipo Cola como la anteriormente descrita.

SE PIDE:

Codificar un método en Java que asigne un determinado elemento (carácter), que se recibe como parámetro, en una determinada posición j también recibida como parámetro.

OBSERVACIONES:

- No se permite la utilización de ninguna estructura de datos auxiliar.
- Sólo se permite la realización de un único recorrido en la cola.
- El procedimiento deberá garantizar que el acceso a una determinada posición es siempre posible.

A continuación se presenta el código de la operación de inicialización del Vector cuya ejecución se supone previamente realizada.

```
final int tamaño = 10;
final char vacio = '·';
public Vector () {
   vector = new tad_cola ();
}
public void inicializar () {
   int i;
   for (i = 1; i<=tamaño; i++)
      vector.encolar (vacio);
}
```

Orientación.

El ejercicio presenta un caso limitado de obtención de un TAD (Vector) a partir de otro (Cola). Se trata de implementar la operación de escribir un dato (*dato*) en una posición (*j*) válida del *vector*.

Dado que se conoce el número de elementos del *vector* se utilizará la técnica iterativa. El tratamiento requiere cuatro fases:

- Inicio. Se verifica que *j* es una posición válida.
- Avance. Se progresa (*desencolar – encolar*) sobre los elementos del *vector* anteriores a *j*.
- "Escritura". Se sustituye (encolar) el valor actual encontrado (desencolado) en la posición *j* por el *dato* suministrado.
- Terminación. Se "recicla" el resto de elementos del *vector* para conservar el orden inicial.

Se ha optado por definir el método (*insertar*) dentro de una clase (*Vector*) donde, también, se incluye la variable miembro *vector (*de la clase Cola de caracteres). Por tanto, el método insertar será un método de objeto.

Código.

```
public void insertar (char dato, int j)  {
    char elem;
    int i;
    if ( (j > tamaño) || (j < 1))
        System.out.println ("Error. Posicion fuera de rango");
    else {
        for (i = 1; i < j; i++) {
            elem = vector.desencolar ();
            vector.encolar (elem);
        }
        elem = vector.desencolar ();
        vector.encolar (dato);
        for (i = 1; i <= (tamaño-j); i++) {
            elem = vector.desencolar ();
            vector.encolar (elem);
        }
    }
}
```