Estructuras de Datos y Algoritmos. Curso 2009/10
Unidad Didáctica II: EDAs Lineales y su Jerarquía Java.

Tema 1. Lista, Pila y Cola: Modelo y Aplicaciones

Mabel Galiano Natividad Prieto

Departamento de Sistemas Informáticos y Computación
Escuela Técnica Superior de Ingeniería Informática

Índice

Objetivos y Bibliografía

El objetivo general de este tema es presentar las Estructuras de Datos Lineales (EDAs) que más frecuentemente se usan en el desarrollo de aplicaciones: Lista Con Punto de Interés (PI), Pila y Cola. En concreto, para subrayar la independencia entre Modelo e Implementación de una EDA en este tema se definirán tan sólo las interfaces Java que representan los Modelos de Lista Con PI, Pila y Cola y se ilustrará su uso reutilizándolas en el desarrollo de algunas aplicaciones de mediana complejidad; será en el próximo tema cuando se presentarán las posibles implementaciones de estas interfaces y se discutirá cuál de ellas resulta más adecuada según el criterio de eficiencia. Esta forma de plantear el diseño de las EDAs Lineales permitirá resaltar una vez más las ventajas que proporciona la Programación Orientada a Objetos: Reutilización y Portabilidad del Software vía Genericidad, Herencia y Ocultación de la Información.

Como bibliografía básica del tema se propone consultar la siguiente: los Capítulos 9 y 10 del texto de Sahni, S. **Data Structures, Algorithms and Applications in Java** (McGraw-Hill Higher Education, 2000) y los apartados 2 y 3 del Capítulo 6 del libro de Weiss M.A. **Estructuras de datos en Java** (Adisson-Wesley, 2000).

1. Características y tipos de las EDAs Lineales

Atendiendo a la clasificación propuesta en el tema `I_6`, se considera que una EDA es Lineal si se emplea para la gestión Secuencial de los Datos de una Colección o, equivalentemente, si su Modelo describe el acceso uno tras otro por orden de inserción a los Datos de una Colección. A su vez, y como se tendrá ocasión de detallar en los siguientes apartados de este tema, las restricciones de acceso a los Datos de una EDA Lineal determinan su tipo y Modelo; así,

- si sólo se puede acceder al último de sus Datos por orden de inserción entonces la EDA es una Pila porque su Modelo es LIFO (*Last In First Out*), o viceversa;
- si sólo se puede acceder al primero de sus Datos por orden de inserción entonces la EDA es una Cola porque su Modelo es FIFO (*First In First Out*), o viceversa;
- si se puede acceder a cualquiera de sus Datos por orden de inserción, tras haber accedido al anterior, entonces la EDA es una Lista porque su Modelo es el General de Acceso Secuencial, o viceversa; en concreto, si se interpreta que el único Dato al que se tiene acceso en una Lista es el que ocupa su Punto de Interés (PI) y que el PI se desplaza secuencialmente del primero al último de los Datos entonces la Lista es una Lista Con PI.

En cualquiera de estos casos, nótese, si no se quiere sobrecargar el coste "natural" de la gestión Secuencial parece claro que cada una de las operaciones de acceso que describe un Modelo Lineal debe tener un coste máximo estimado estrictamente del orden de una constante.

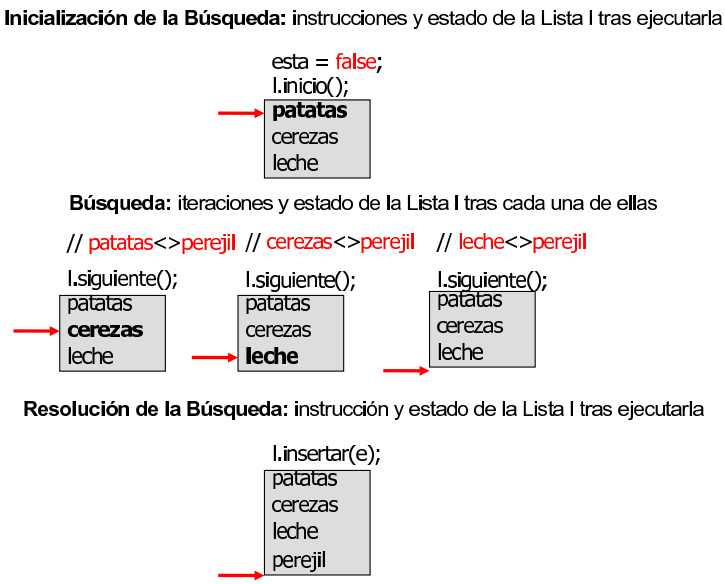
2. Lista Con Punto de Interés

Como ya se sabe, el Acceso Secuencial a una Colección se realiza visitando uno tras otro por orden de inserción todos y cada uno de los Datos que la componen. Un ejemplo familiar de este acceso es el que se realiza para manipular la Lista de la Compra, en general no ordenada y cuyos Datos son nombres de productos: típicamente, para confeccionarla se busca en primer lugar el nombre de un producto determinado, por si ya figura, haciendo un Recorrido Secuencial Ascendente de ésta desde su inicio vía operación `siguiente()`; si se alcanza el fin de la Lista sin haberlo encontrado entonces lo normal es añadir el nuevo nombre al final de la Lista, tras el último. Ya en general, cualquier Exploración (Recorrido o Búsqueda) Secuencial Ascendente de una Colección se debe expresar mediante una Composición Secuencial de alguna de las siguientes operaciones: `inicio()`, que inicializa el Acceso Secuencial a la Colección a partir de su primer Dato; `recuperar()`, que obtiene el (único) Dato que es accesible en cada momento y `eliminar()` que lo borra; `siguiente()`, que hace accesible el siguiente Dato de la Colección; `esFin()`, que comprueba si quedan más Datos a los que acceder; `insertar(e)` que añade un nuevo Dato `e` a la Lista; `esVacia()`, que comprueba si la Lista está o no vacía; volviendo al ejemplo de la Lista de la Compra, en términos de las operaciones básicas definidas la confección de una cierta Lista `l` se expresaría mediante una Búsqueda Ascendente como la siguiente:

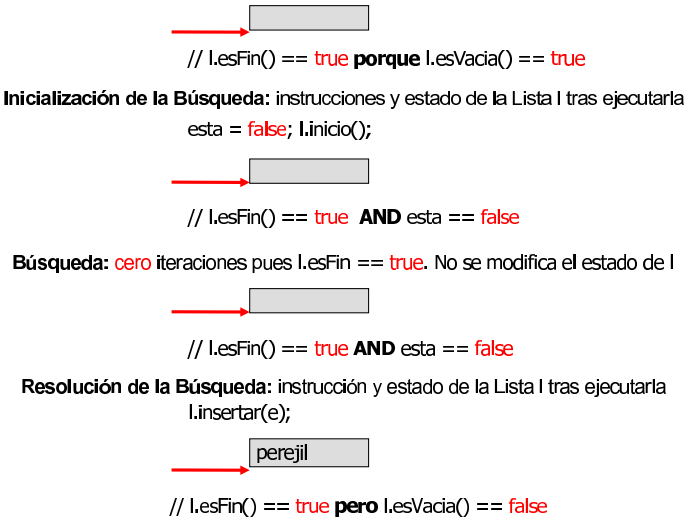
```
l.inicio();
while ( !l.esFin() && !l.recuperar().equals(e) ) l.siguiente();
if ( l.esFin() ) l.insertar(e);
```

Nótese que en cada paso de esta Búsqueda cualquier operación que sufre la Lista `l` afecta **implícitamente** al único de sus Dato accesible en tal paso, el que ocupa **el Punto de Interés** (PI); la siguiente figura lo ilustra mostrando los sucesivos estados de una Lista sobre la que se

busca el producto `perejil`, representando cada uno de ellos como un recuadro sombreado que encierra los Datos de la Lista junto con la flecha que marca su PI:



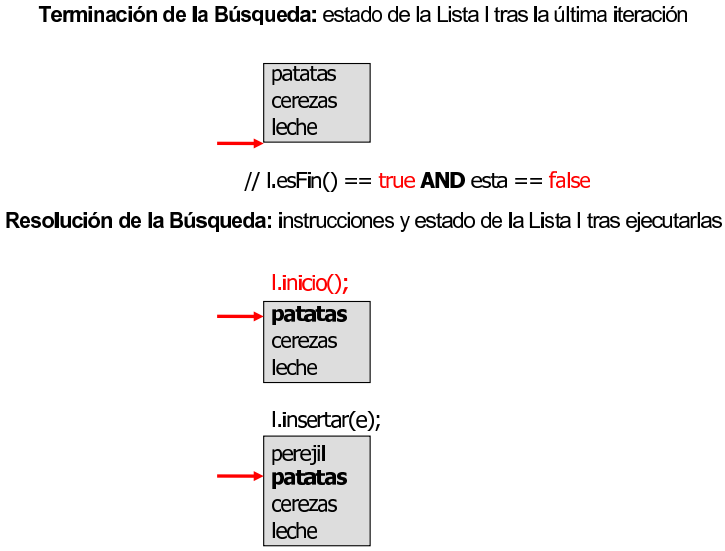
Obsérvese que la ejecución de `l.inicio()` sitúa el PI de la Lista sobre su primer Dato, independientemente del lugar que ocupara antes; a partir de aquí, mientras se cumpla la guarda del bucle, en cada iteración se procederá a recuperar el Dato al que apunta el PI actual (la flecha) mediante la instrucción `l.recuperar()` y si no es igual al Dato `e` buscado se avanzará el PI ejecutando `l.siguiente()`; así hasta que bien se encuentre `e` o bien, como en la figura, el PI se sitúe tras el último Dato de la Lista, condición de terminación que se evalúa aplicando `esFin()` a la Lista `l`. Una vez alcanzado el fin de Lista la resolución de la Búsqueda mostrada en la figura, `l.insertar(e)`, permite observar un detalle relevante de la semántica de la inserción en una Lista Con PI: **el nuevo Dato siempre se inserta antes del que ocupa el PI**, de donde tras su inserción `perejil` es el último Dato de la Lista, el PI queda inalterado como antes de la inserción y se mantiene la condición `l.esFin()`. Nótese también que si la Lista de la Compra está inicialmente vacía, la ejecución de la Búsqueda propuesta provoca la inserción de su primer Dato también al final de la Lista, como se observa en la siguiente figura:



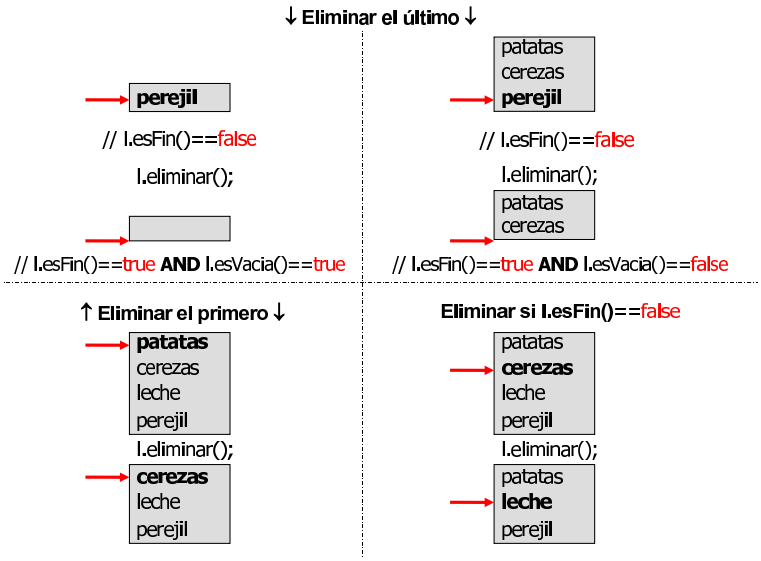
Obviamente, la semántica de inserción descrita es independiente del lugar donde se encuentre el PI de una Lista; así, para insertar `e` al inicio de la Lista en vez de al final basta con modificar como sigue la resolución de la Búsqueda propuesta:

```
l.inicio();
while ( !l.esFin() && !l.recuperar().equals(e) ) l.siguiente();
if ( l.esFin() ){ l.inicio(); l.insertar(e); }
```

En efecto, y como muestra la figura siguiente, al ejecutar `l.inicio()` antes que `l.insertar(e)` se consigue que `e` sea el primer Dato de la Lista sin alterar su PI -insertar en inicio de Lista- porque siempre se inserta antes del Dato que ocupa el PI actual.



Al igual que la inserción, la operación `eliminar()` merece especial atención en lo que respecta al estado en el que queda el PI de la Lista tras ejecutarla: como permanece inalterado, el PI apunta al Dato siguiente al eliminado, a ninguno si el que se ha borrado es el último de la Lista. En la siguiente figura se plantean varias situaciones que ilustran esta semántica:



En base a la introducción y ejemplos realizados, se propone la siguiente definición:

> una EDA **Lista Con Punto de Interés** (PI) es una Colección Homogénea de Datos que se gestionan siguiendo un criterio **Secuencial**, i.e. uno tras otro desde el primero al último en orden de inserción; el Dato de la Lista que resulta accesible en cada momento de esta gestión es entonces el que ocupa su Punto de Interés.

De esta definición se desprende que si no hay Dato alguno bajo el PI, bien porque la Lista está vacía (`esVacia() == true`) o porque el Acceso a ella ha concluido (`esFin() == true`), la ejecución de las operaciones `recuperar()`, `eliminar()` y `siguiente()` provoca un error de ejecución que se debe prevenir estableciendo `!esFin()` como Precondición para su uso; así mismo, hacer notar que la gestión de todos los Datos de una Lista Con PI es posible ya que las operaciones `inicio()` y `siguiente()` permiten modificar el Punto de Interés actual.

Ya para concluir con la introducción de la Lista Con PI conviene realizar dos consideraciones más. En primer lugar puntualizar que para mejorar la eficiencia de determinadas operaciones de Recorrido y Búsqueda sobre una Lista Con PI, como la concatenación de una Lista con otra dada, conviene que su Modelo incluya también la operación modificadora `fin()`, que desplaza directamente el PI al final de la Lista. Así mismo resulta de interés señalar el motivo pro el que el Modelo de esta EDA resulta tan familiar: porque es el único que se ha utilizado hasta la fecha en PRG y en esta asignatura **aunque implícitamente**, esto es a través de sus Implementaciones Contigua o Enlazada; para resaltar este hecho, en el siguiente ejemplo se presentan tres codificaciones alternativas del método `toString()` de una Lista con PI: la primera utiliza las operaciones del Modelo Lista Con PI que se acaba de presentar, la segunda las de su Representación sobre un `array` y la tercera las de su Representación sobre una `LEG`.

```
/** Recorrido de una Lista Con PI **/
String res = "";
for ( this.inicio(); !this.esFin(); this.siguiente() )
    res += this.recuperar().toString()+" ";

/** Recorrido del array que representa una Lista Con PI **/
String res = "";
for ( int i = 0; i < this.talla; i++ )
    res += this.array[i].toString()+" ";

/** Recorrido de la LEG que representa una Lista Con PI **/
String res = "";
for ( NodoLEG<E> aux = this.primero; aux != null; aux = aux.siguiente )
    res += aux.dato.toString()+" ";
```

Obsérvese que mientras en el código del Recorrido del `array` y la `LEG` el Punto de Interés se representa explícitamente mediante una variable, el índice `i` del `array` o el Enlace `aux` de la `LEG`, en el del Modelo Lista Con Punto de Interés **no**, es un parámetro implícito del que la representación específica se relega a su Implementación.

2.1. Modelo Java: la interfaz genérica `ListaConPI`

Se presenta a continuación la interfaz genérica `ListaConPI` que representa en Java el Modelo de una Lista Con Punto de Interés (PI) antes descrito; específicamente, los métodos que lo componen se presentan clasificados por el tipo de operación que definen (consultora o modificadora de la Lista o de su PI) y con los comentarios necesarios para especificarlos:

```
package librerias.estructurasDeDatos.modelos;
/** ListaConPI<E>: Base de la Jerarquía que representa el Modelo de una Lista Con PI
  * de Elementos de tipo E, o describe en Java las operaciones de una Lista Con PI
  **/
public interface ListaConPI<E> {
// métodos Modificadores del estado de una Lista Con PI:
  /** inserta el Elemento e en una Lista ANTES del que ocupa su PI,
    * que permanece inalterado **/
  void insertar(E e);
  /** SII !esFin(): elimina de la Lista el Elemento que ocupa su PI,
    * que permanece inalterado **/
  void eliminar();
// métodos Modificadores del estado del PI de una Lista:
  /** sitúa el PI de una Lista en su inicio **/
  void inicio();
  /** SII !esFin(): avanza el PI de una Lista **/
  void siguiente();
  /** sitúa el PI de una Lista en su fin **/
  void fin();
// métodos Consultores del estado de una Lista Con PI:
  /** SII !esFin(): obtiene el Elemento que ocupa el PI de una Lista **/
  E recuperar();
  /** comprueba si el PI de una Lista está en su fin **/
  boolean esFin();
  /** comprueba si una Lista Con PI está vacía **/
  boolean esVacia();
}
```

Ejercicios propuestos:

1. Una versión alternativa de la interfaz `ListaConPI` presentada sería la que se contempla el lanzamiento de la Excepción de Usuario `AccesoIncorrecto` cuando se intentan utilizar de forma incorrecta sus métodos `recuperar()`, `siguiente()` y `eliminar()`. Defínase esta nueva interfaz, con nombre `ListaConPIYExcepciones`, y discútanse las diferencias que conlleva el uso de los métodos de uno y otro Modelo Java de Lista Con PI.

2. Con los métodos de la interfaz `ListaConPI` propuesta solo se puede realizar un Acceso Secuencial **Ascendente**. Con el fin de posibilitar el **Descendente**, ¿qué métodos sería necesario añadir a la interfaz actual? Determínense sus perfiles y constrúyase una nueva interfaz `ListaConPIDown` que represente el Modelo de Acceso Secuencial Descendente.

3. Determínese la composición secuencial de métodos de `ListaConPI` que construye, a partir de una vacía, la Lista Con PI de `Integer 1, 2, 3, 5, 6`. Descríbase además gráficamente los estados de la Lista según se ejecute cada una de las instrucciones propuestas.

4. Dada la Lista de `Integer 1, 2, 3, 5, 6` cuyo Punto de Interés está situado sobre el `6`, discútase si es equivalente ejecutar «`insertar(new Integer(4)); eliminar();`» que «`eliminar(); insertar(new Integer(4));`»

2.2. Ejemplos de uso

Como ya se ha indicado, una interfaz Java como `ListaConPI` se puede reutilizar en el diseño de nuevas clases que tienen como atributo un objeto de tipo Estático `ListaConPI`. Así por ejemplo, supóngase que se debe desarrollar la Implementación de la interfaz `ColaPrioridad` presentada en el tema `I_6` y que para ello sólo se dispone de la interfaz `ListaConPI` y de su implementación Enlazada `LEGListaConPI`, de la que además del nombre sólo se conoce su ubicación; en estas condiciones la implementación Lineal de `ColaPrioridad` que se obtendría usando vía Composición la Jerarquía de `ListaConPI` sería la siguiente:

```
// ubicación de cualquier Implementación Lineal de una EDA, como LEGListaConPI
package librerias.estructurasDeDatos.lineales;
// ubicación de cualquier Modelo Java de una EDA
import librerias.estructurasDeDatos.modelos.*;
public class LPIColaPrioridad<E extends Comparable<E>> implements ColaPrioridad<E> {
  // para representar la Cola de Prioridad se usa la ListaConPI l
  protected ListaConPI<E> l;
  /** crea una Cola de Prioridad (CP) vacía, i.e. crea vacía la ListaConPI l **/
  public LPIColaPrioridad(){ this.l = new LEGListaConPI<E>(); }
  /** inserta el Elemento e en una Cola de Prioridad,
    * o inserta ordenadamente e en la ListaConPI l **/
  public void insertar(E e){
      this.l.inicio();
      while ( !this.l.esFin() && this.l.recuperar().compareTo(e) <= 0 )
          this.l.siguiente();
      this.l.insertar(e);
  }
  /** SII !esVacia(): obtiene el Elemento con máxima prioridad de una CP,
    * u obtiene el primer Elemento de la ListaConPI l **/
  public E recuperarMin(){ this.l.inicio(); return this.l.recuperar(); }
  /** SII !esVacia(): obtiene y elimina el Elemento con máxima prioridad de una CP,
    *  u obtiene y elimina el primer Elemento de la ListaConPI l**/
  public E eliminarMin(){...}
  /** comprueba si una CP está vacía, o comprueba si la ListaConPI l es vacía **/
  public boolean esVacia(){...}
  /** obtiene el String con los Elementos de una CP, o el toString de la ListaConPI l **/
  public String toString(){...}
}
```

Nótese que si los métodos de la interfaz `ListaConPI` se implementan con un coste Temporal constante, entonces el coste del método `insertar` de `LPIColaPrioridad` es lineal con la talla de la Cola de Prioridad y el del resto de los métodos es constante.

Ejercicios propuestos:

1. Complétese el diseño de la clase `LPIColaPrioridad`.
2. Sea una `ListaConPI` de `Integer`, potencialmente vacía. Se pide diseñar un método que, sin hacer uso de una Lista auxiliar, modifique sus Datos de forma que cada uno de ellos pase a valer el triple de su valor original. Por ejemplo, si inicialmente se tiene la Lista de `Integer` `3, 121, -6, 5, 4, 5, 22, -1` tras la modificación requerida la Lista pasa a ser `9, 363, -18, 15, 12, 15, 66, -3`.

3. Sea la siguiente una subinterfaz de `ListaConPI`:

```
package librerias.estructurasDeDatos.modelos;
import librerias.excepciones.*;
public interface ListaConPIPlus<E> extends ListaConPI<E> {
    /** obtiene la talla de una Lista Con PI **/
    int talla();
    /** comprueba si el Elemento e está en una Lista Con PI **/
    boolean contiene(E e);
    /** elimina la primera aparición del Elemento e en una Lista Con PI
      * y devuelve true, o devuelve false si e no está en la Lista **/
    boolean eliminar(E e);
    /** si el Elemento e está en una Lista Con PI elimina su última
      * aparición y la devuelve como resultado; sino lo advierte
      * lanzando la Excepción ElementoNoEncontrado **/
    E eliminarUltimo(E e) throws ElementoNoEncontrado;
    /** si el Elemento e está en una Lista Con PI elimina todas sus
     *  apariciones en ella; sino lo advierte lanzando ElementoNoEncontrado **/
    void eliminarTodos(E e) throws ElementoNoEncontrado;
    /** elimina todos los Datos de una Lista Con PI **/
    void vaciar();
    /** concatena una Lista Con PI con otra **/
    void concatenar(ListaConPI<E> otra);
    /** invierte in-situ una Lista a partir de su PI **/
    void invertirDesdePI();
}
```

La clase `LEGListaConPIPlusAlumnos`, disponible en la tabla de material didáctico auxiliar de los Contenidos PoliformaT del tema, es una Implementación de esta subinterfaz cuyos métodos bien contienen errores o bien están incompletos. Se pide corregirla y completarla, indicando también el coste Temporal de cada uno de sus métodos.

Nota: en la tabla de material didáctico auxiliar de los Contenidos PoliformaT del tema también figuran dos ficheros que permiten comprobar la correcta realización del ejercicio: el programa `TestListaConPIPlus.java` y el fichero `TestCorrectoListaConPIPlus.txt`; la ejecución del primero coincidirá línea a línea con el contenido del segundo si las modificaciones realizadas en `LEGListaConPIPlusAlumnos` son las pertinentes.

4. Si `ListaConPIPlus` extendiera la interfaz `ListaConPIYExcepciones`, ¿qué tipo de modificaciones se deberían realizar en su código? ¿Y en el de `LEGListaConPIPlusAlumnos`?

3. Pila

La idéntica organización y manipulación de Colecciones de Datos tan dispares como las Bandejas de un autoservicio de comedor, las Hojas de papel de la bandeja de la impresora, los Platos del fregadero de una cocina o los Periódicos de los expositores de un kiosco se describe comúnmente mediante el término **Pila**, que también presupone o comporta realizar la inserción, borrado y recuperación de sus Datos **siempre** por un mismo punto, su **tope** y, además, denominar a dichas operaciones `apilar`, `desapilar` y `tope` respectivamente.

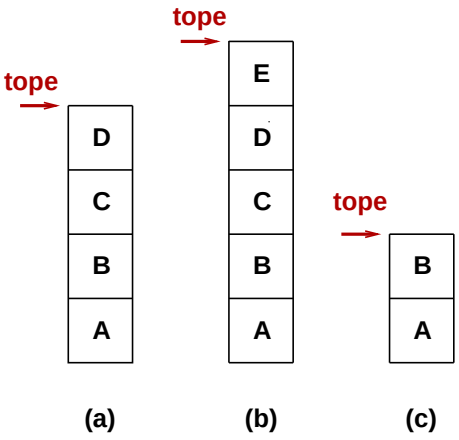
La gestión de Datos de tipo **LIFO** (***L**ast **I**n **F**irst **O**ut*) descrita no sólo se observa en múltiples situaciones de la vida real como las descritas sino que además, o quizás por ello, se traslada

al ámbito de la Computación, donde junto con la Cola es una de las EDAs más utilizadas; por ejemplo, recuérdese que en el tema `I_5` se denominó **Pila de la Recursión** a la estructura que soporta la ejecución de un método recursivo y está compuesta por Registros de Activación; en efecto, cada vez que se produce una nueva llamada o invocación a un método recursivo, se crea el correspondiente Registro de Activación `r` con la información asociada a dicha llamada y se añade a la Pila actual, esto es se ejecuta una operación `apilar` que sitúa a `r` en el tope de la Pila; además, como en todo momento la información disponible es la que ocupa el tope de la Pila, bastará ejecutar la operación `tope()` para recuperarlo y la operación `desapilar()` para borrarlo cuando la llamada termine.

En base a la introducción que se acaba de realizar se propone la siguiente definición:

> una EDA **Pila** es una Colección Homogénea de Datos que se gestionan siguiendo un criterio **LIFO**, esto es permitiendo únicamente el acceso al último de sus Datos en orden de inserción.

El siguiente ejemplo muestra gráficamente el comportamiento de esta EDA Lineal:



Inicialmente, figura (a), se dispone de una Pila de cuatro `Character` cuyo tope ocupa `D`; si a esta Pila se le añade un nuevo Dato ejecutando «`apilar(new Character('E'));`» el Carácter `E` se situará en su tope, como muestra la figura (b); si ahora se ejecuta «`desapilar();`» se borrará de la Pila el Carácter `E` que ocupa su tope; si se borran dos Datos más se obtiene la Pila de la figura (c), en cuyo tope se encuentra el Carácter `B`.

Ya para concluir es importante subrayar que con el tope de una Pila ocurre algo muy similar a lo que ya se indicó para el Punto de Interés de una Lista, lo que no es de extrañar pues ambos son los únicos puntos de acceso efectivo de sus respectivas EDA: no aparece explícitamente como parámetro de ninguna de las operaciones que definen la gestión LIFO, por lo que su representación específica queda relegada a la fase de Implementación; cuando no hay Dato alguno que lo ocupe, pues la Pila está vacía (`esVacia() == true`), la ejecución de las operaciones `tope()` y `desapilar()` provocará un error que se debe prevenir usándolas con la Precondición `!esVacia()`. Finalmente, notar que aunque sólo se tiene acceso al Dato que ocupa el tope de una Pila, la operación `desapilar()` permite modificarlo y, por tanto, manipular todos sus Datos.

3.1. Modelo Java: la interfaz genérica `Pila`

El sencillo Modelo que definen las cuatro operaciones básicas de manejo de una Pila se representa en Java mediante la siguiente interfaz:

```
package librerias.estructurasDeDatos.modelos;
/**Pila<E>: Base de la Jerarquía que representa el Modelo de una Pila de
 * Elementos de tipo E, o describe en Java las operaciones de una Pila **/
public interface Pila<E> {
// métodos Modificadores del estado de una Pila:
  /** inserta el Elemento e en una Pila, o lo sitúa en su tope **/
  void apilar(E e);
  /** SII !esVacia(): obtiene y elimina de una Pila
    *  el Elemento que ocupa su tope **/
  E desapilar();
// métodos Consultores del estado de una Pila:
  /** SII !esVacia(): obtiene el Elemento que ocupa el tope de una Pila **/
  E tope();
  /** comprueba si una Pila está vacía **/
  boolean esVacia();
}
```

Sobre esta interfaz conviene puntualizar que, con el fin de lograr un uso más cómodo de los métodos de `Pila`, se ha sobrecargado la especificación del método `desapilar()` para que realice dos acciones: la consultora equivalente a la del método `tope()` y la modificadora de eliminación del Dato que ocupa el tope de la Pila.

3.2. Ejemplos de uso

A continuación se presentan dos problemas en cuya estrategia de resolución interviene una Pila; con la realización de su diseño en Java se pretende incidir una vez más en las ventajas que conlleva la Reutilización del Software a la hora de programar y, específicamente, mostrar que un buen diseñador debe ser capaz de construir una gran cantidad de nuevos métodos a partir del reducido grupo de básicos que figuran en una interfaz, como `Pila` en este caso; obviamente, para poder ejecutar los diseños realizados se debe disponer además, **tan solo**, del nombre de una implementación de la interfaz y el del paquete que la contiene, supóngase `ArrayPila` del paquete `librerias.estructurasDeDatos.lineales` para los diseños posteriores.

El problema de los paréntesis

Dada una expresión parentizada se quiere diseñar un método Java que establezca la correspondencia entre los paréntesis abiertos y cerrados que aparecen en ella; específicamente, como resultado de su ejecución se deberán mostrar por pantalla los pares de paréntesis (abiertos y cerrados) de la expresión que sí se corresponden y, si existen, los paréntesis sin pareja. Por ejemplo, si la expresión parentizada (`a*(b+c)+d`) se representa mediante un `String e`, el resultado del método sería el que figura a continuación:

```
Introduzca la expresión parentizada a analizar: (a*(b+c)+d)
El paréntesis abierto en posición 3 se corresponde con el cerrado de la 7
El paréntesis abierto en posición 0 se corresponde con el cerrado de la 10
```

Si por el contrario la expresión parentizada es (a+b)), el resultado sería:

```
Introduzca la expresión parentizada a analizar: (a+b))
El paréntesis abierto en posición 0 se corresponde con el cerrado de la 4
Sin pareja el paréntesis cerrado de la posición 5
```

La clave para resolver el problema es observar que durante un análisis de izquierda a derecha de la expresión cada paréntesis cerrado que se detecte debe corresponderse con el último abierto ya analizado. Nótese que para recordar cuál fué el último paréntesis abierto que se analizó es necesario disponer, además de la expresión a analizar, de una memoria auxiliar donde apuntarlo tan pronto se encuentre y de donde extraer la información (recordarlo) cuando sea necesario, esto es cuando se detecte un paréntesis cerrado; esta memoria auxiliar es en Programación ni más ni menos que una Pila.

A partir de lo anterior, la estrategia de diseño a seguir es un Recorrido Ascendente de `e` en el que el tratamiento a realizar sobre cada una de sus componentes, `e.charAt(i)`, $0 \leq$ `i` $<$ `e.length`, consiste en comprobar si se trata de un paréntesis abierto, cerrado o cualquier otro carácter y, según sea, realizar una de las siguientes acciones: si `e.charAt(i)=='('`, apuntar su posición `i` en la Pila `p` que representa la memoria auxiliar de la que se dispone, para después poderla recordar, esto es `p.apilar(new Integer(i))`; sino, si `e.charAt(i)==')'` y la Pila `p` está vacía mostrar por pantalla la posición `i` del paréntesis cerrado sin pareja; sino, si `e.charAt(i)==')'` **pero** `!p.esVacia()` entonces mostrar por pantalla la correspondencia encontrada y eliminar el paréntesis abierto de la Pila (`p.desapilar()`); finalmente, si `e.charAt(i)` no es un paréntesis entonces pasar a la siguiente (`i++`). Al terminar el Recorrido, si la Pila `p` no está vacía es que hay paréntesis abiertos sin emparejar, que se deberán mostrar por pantalla indicando la posición que ocupan. La transcripción a Java de la solución propuesta sería el siguiente programa:

```
package ejemplos.temaII_1;
import librerias.util.gestionES.*;
import librerias.estructurasDeDatos.modelos.*;
import librerias.estructurasDeDatos.lineales.*;
public class TestParentesis {
 public static void main(String args[]){
     GestorES ioDevice = new ConsolaGestorES();
     String expresion = ioDevice.getString("Introduzca la expresión a analizar:");
     ioDevice.escribir(correspondenciaParentesis(expresion));
 }
 private static String correspondenciaParentesis(String e){
     String res = ""; int talla = e.length();
     Pila<Integer> p = new ArrayPila<Integer>();
     for ( int i = 0; i < talla; i++ )
         if ( e.charAt(i) == '(' ) p.apilar(new Integer(i));
         else if ( e.charAt(i) == ')' )
              if ( !p.esVacia() ){
                res += "El paréntesis abierto en posición "+p.desapilar();
                res += " se corresponde con el cerrado de la "+i+"\n";
              }
         else res += "Sin pareja el paréntesis cerrado de la posición "+i;
     while ( !p.esVacia() )
         res += "Sin pareja el paréntesis abierto de la posición "+p.desapilar();
     return res;
 }
}
```

Obsérvese que el método `correspondenciaParentesis` presentado tendrá un coste Temporal Asintótico lineal con la talla del problema `e.length` **siempre que** `ArrayPila` **implemente en tiempo constante los métodos de la interfaz** `Pila`.

Ejercicios propuestos:

1. En el problema de las Torres de Hanoi para mover un disco de una torre a otra se realiza una gestión LIFO, por lo que cada torre se podría representar como una Pila. Escríbase un método recursivo en Java que contemple este nuevo planteamiento y muestre por pantalla los diferentes estados en los que se encuentran las tres torres durante su ejecución.

2. Diséñese un programa Java que tras inicializar desde teclado una Pila de `Integer` la muestre por pantalla, obtenga recursivamente una nueva Pila con los mismos Datos que la original pero cambiados de signo y la muestre por pantalla.

3. Se dice que una Pila dada `p` es sombrero de otra si todos los Datos de `p` aparecen en la otra en el mismo orden y ocupando las posiciones más próximas a su tope; por ejemplo, si `p` es una Pila vacía o igual a otra seguro que es sombrero de otra. Enriquézcase la interfaz `Pila` con un método recursivo y no estático `esSombrero` que lo compruebe y, además, estímese su coste Temporal Asintótico suponiendo que los métodos de la interfaz `Pila` han sido implementados para ejecutarse en tiempo constante.

4. Estúdiese la conveniencia de definir la talla de una Pila como un método enriquecedor de la interfaz `Pila`.

El Problema del Laberinto

Según la leyenda griega, Ariadna, hija de Minos, entregó a Teseo un ovillo de hilo con el que pudo desandar sus pasos en el Laberinto y una espada para matar al Minotauro; si cual moderno Teseo se pretende desarrollar una aplicación que resuelva el Problema del Laberinto, una Pila servirá a modo de hilo de Ariadna, como se explica en lo que sigue.

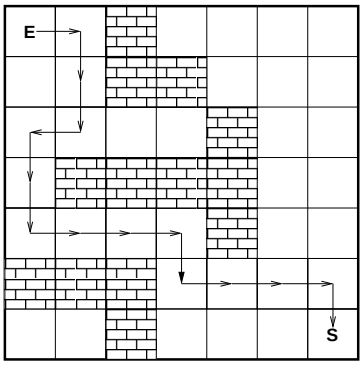
Para encontrar, si existe, un Camino `c` que une los Puntos de entrada `E` y salida `S` de un Laberinto es necesario buscar sistemáticamente, de forma ordenada, el siguiente Punto (`puntoSiguiente`) libre entre todos los que se puedan alcanzar desde el Punto actual de exploración (`puntoActual`, que inicialmente es `E`); si en un momento dado `puntoActual` está rodeado de obstáculos, habrá que volver hacia atrás desandando los pasos dados hasta alcanzar aquel Punto del Camino desde el que aún sea posible seguir explorando otros. Como es fácil deducir, la exploración termina bien cuando `puntoActual` coincide con `S`, el Punto donde concluye cualquier Camino del Laberinto, o bien cuando al volver hacia atrás no hay Punto del Camino alguno desde el que seguir explorando, de donde se deduce que no hay Camino en el Laberinto.

La estrategia de exploración expuesta se puede expresar algorítmicamente mediante el siguiente Esquema de Búsqueda anidada, esto es de Búsqueda de `puntoSiguiente` dentro de la Búsqueda del Camino solución `c`:

```
c.inicializar("Ninguno");
/** Inicialización de la Búsqueda del Camino solución c: si existe, el Punto
  *  de partida de cualquier Camino del Laberinto es E; la exploración se
  *  inicia en él y, por tanto, hay un Camino en construcción **/
Punto puntoActual = E; boolean hayCamino = true;
/** Búsqueda del Camino solución c: SII existe un Camino en construcción y
  *  puntoActual no es su Punto final S, buscar el primer puntoSiguiente libre **/
while ( hayCamino AND puntoActual != S ) {

  /** Búsqueda del primer puntoSiguiente libre: desde puntoActual se aplican
    * las restricciones del problema (desplazamientos permitidos y obstáculos) **/
  Punto puntoSiguiente = null; boolean siguienteLibre = false;
  while ( puntoActual.restanDesplazamientos() && !siguienteLibre ){
     puntoSiguiente = puntoActual.desplazar();
     if ( puntoSiguiente.libre() ) siguienteLibre = true;
  }
  /** Resolución de la Búsqueda del primer puntoSiguiente libre: */
    * si puntoSiguiente es un Punto libre, puntoActual es el último Punto de c;
    * sino, volver hacia atrás. En cualquier caso, puntoSiguiente será el nuevo
    * Punto de exploración en la siguiente iteración **/
  if ( siguienteLibre ) c.actualizarCon(puntoActual);
  else if (!c.esVacio()) puntoSiguiente = c.volverAtras(puntoActual); else hayCamino = false;
  puntoActual = puntoSiguiente;
}
/** Resolución de la Búsqueda del Camino solución c: **/
if ( hayCamino ) c.actualizarCon(S);
return c;
```

Como se puede observar este esquema dista mucho de estar completo; aplicarlo a un ejemplo concreto servirá para detallarlo y, con ello, dilucidar el papel que en él juega una Pila. Así, sea el Laberinto representado por la cuadrícula 7×7 de la siguiente figura, cuyas casillas enladrilladas son obstáculos, las blancas Puntos libres y la línea que une su entrada `E` con su salida `S` un Camino solución `c`:



Suponiendo que desde un Punto dado de dicho Laberinto el siguiente Punto alcanzable es el Punto contiguo situado, en este orden, bien a su derecha (→) o debajo de él (↓) o a su izquierda (←) o arriba de él (↑), la traza del bucle de Búsqueda anidada propuesto sería:

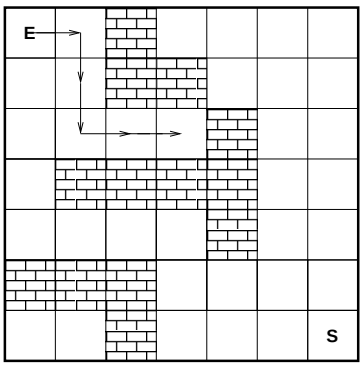
- **Inicialización de la Búsqueda de `c`:** al empezar la Búsqueda su primer Punto debe de ser el de entrada al Laberinto `E`, de donde `puntoActual = E`.
- **Primera iteración de la Búsqueda de `c`:** tras comprobar que se cumple su guarda se realiza **una** sóla iteración del bucle de Búsqueda del primer `puntoSiguiente` libre desde `puntoActual = E`; en efecto, como se puede observar en el dibujo del Laberinto, la casilla contigua a la derecha de `E`, que se notará como `E`→, está libre. Por tanto, siguiendo el esquema se confirma `E` como el último Punto del Camino `c` en construcción, `c.actualizarCon(puntoActual) = E`, y además se actualiza el valor de `puntoActual` con el de `puntoSiguiente` para la siguiente iteración.
- **Segunda Iteración de la Búsqueda de `c`:** como `puntoActual == E→ AND hayCamino` la guarda del bucle se cumple y comienza la Búsqueda del primer `puntoSiguiente` libre desde `puntoActual`; como el Punto contiguo a la derecha de `E`→ es un obstáculo, se necesitan **dos** iteraciones para resolverla con éxito, esto es para establecer que el primer `puntoSiguiente` libre a partir de `puntoActual` es `E`→↓, el que está situado justo debajo de él. Se añade entonces a `c` `puntoActual`, i.e. `c = E E`→, y se actualiza `puntoActual` al valor de `puntoSiguiente`.

  Como se propone comprobar al alumno, hasta concluir la quinta iteración la traza continua por los mismos derroteros y con los siguientes resultados:

  Tercera Iteración: se añade `E`→↓ como último Punto de `c` y se actualiza `puntoActual` a `puntoSiguiente = E`→↓↓.

  Cuarta Iteración: se añade `E`→↓↓ como último Punto de `c` y se actualiza `puntoActual` a `puntoSiguiente = E`→↓↓→.

  Quinta Iteración: se añade `E`→↓↓→ como último Punto de `c` y se actualiza `puntoActual` a `puntoSiguiente = E`→↓↓→→. En la siguiente imagen se muestra el estado del laberinto y de `c` antes de comenzar la sexta iteración.



- **Sexta Iteración de la Búsqueda de `c`:** esta iteración es la primera donde la Búsqueda del primer `puntoSiguiente` libre no tiene éxito, pues `puntoActual = E`→↓↓→→ está rodeado de obstáculos excepto por su izquierda, pero tampoco este desplazamiento está permitido pues, nótese, `puntoActual` se alcanza desde el último Punto del Camino `E`→↓↓→ utilizando un desplazamiento →; para dar cuenta de ello en el esquema propuesto, basta con incluir en él una instrucción que marque como obstáculo cada primer `puntoSiguiente` libre que se encuentre:

```
...
if ( siguienteLibre ) {
    c.actualizarCon(puntoActual);
    puntoSiguiente.marcarObstaculo();
}
else if ( !c.esVacio() ) puntoSiguiente = c.volverAtras(puntoActual);
else hayCamino = false;
...
```

Además de mostrar la necesidad de marcar los Puntos libres que forman parte de `c`, el hecho de que `puntoActual` esté rodeado de obstáculos obliga a analizar en qué consiste exactamente la instrucción `c.volverAtras(puntoActual)` cuando `!c.esVacio()`. Conceptualmente su función es clara: cuando `puntoActual` esté rodeado de obstáculos, i.e. no sea un Punto de `c`, se debe volver hacia atrás desandando los pasos dados hasta alcanzar aquel Punto de `c` desde el que sí sea posible aún seguir explorando otros; por ejemplo, y suponiendo que se dispone de toda la historia de la traza realizada hasta alcanzar `puntoActual = E→↓↓→→`, de su aplicación resulta que para seguir construyendo el Camino solución hay que **desandar los dos últimos pasos del Camino `c` construido**, esto es **volver al Punto `E→↓↓` desde el que se inició ¡la cuarta iteración!**; nótese que, además, una vez recuperado este Punto de `c` habrá que desplazarse desde él al Punto contiguo situado a su izquierda (el contiguo a su derecha ya ha sido explorado sin éxito).

Ahora bien, y éste es el aspecto más importante del análisis a realizar, para poder recordar el estado de la Búsqueda más de una iteración atrás es necesario disponer de una memoria auxiliar sobre la que realizar, básicamente, dos acciones: apuntar en ella el Punto del Camino `c` que se determine en cada iteración, que sería `c.actualizarCon(puntoActual)`, y consultar y eliminar su último Dato para usarlo como `puntoSiguiente` cuando en una iteración posterior `puntoActual` esté rodeado de obstáculos, que si `!c.esVacio()` sería `c.volverAtras(puntoActual)`. Si esta memoria auxiliar se representa con una Pila `p`, entonces: sus Datos son los Post-It utilizados para apuntar el último Punto del Camino establecido en cada iteración y el siguiente desplazamiento permitido desde dicho Punto, por si en una iteración posterior se volviera a realizar la exploración desde él; `apilar` en `p` el Post-It confeccionado a partir del `puntoActual` de la iteración es `c.actualizarCon(puntoActual)` y «`if (!p.esVacia()) puntoSiguiente=p.desapilar()`» representa la acción `c.volverAtras(puntoActual)` si `!c.esVacio()`, de donde si `p` está vacía la ejecución de `hayCamino = false` concluye con la iteración y la Búsqueda de `c`; finalmente hacer notar que en cualquier iteración de la Búsqueda de `c`, la Pila `p` contiene todos los Puntos del Camino en construcción, por lo que si concluye con éxito `hayCamino`, `puntoActual = S`, y tras apilar `S`, `c = p.toString()` permite obtener su resultado.

Volviendo ahora a la traza de la sexta iteración, y suponiendo actualizada convenientemente la Pila de Post-It `p`, al resolver sin éxito la Búsqueda del primer `puntoSiguiente` libre alcanzable desde `puntoActual = E→↓↓→→`, se considerará que `puntoSiguiente` es el último Punto del Camino que figura en el tope de `p`, esto es la primera componente del Post-It `(E→↓↓→, ↓)` apilado en la quinta iteración -recuérdese que ↓ es el siguiente desplazamiento que desde él se puede realizar, pues → ha sido el utilizado para establecer el `puntoActual` de la sexta iteración. Nótese además que al desapilar este Post-It el

último Punto del Camino en construcción es ahora E→↓↓, lo que supone desandar uno de los dos pasos del Camino c dados incorrectamente y, por ello, permitir que continue la Búsqueda del Camino solución.

- **Séptima Iteración de la Búsqueda de c:** de nuevo puntoActual = E→↓↓→ está rodeado de obstáculos excepto por su izquierda, lo que tampoco sirve ya que ha sido alcanzado desde el último punto del Camino E→↓↓ utilizando un desplazamiento →. La situación es exactamente la misma que en la anterior iteración solo que un instante después, cuando aún queda por desandar un paso. Por tanto, se tomarán las mismas decisiones que entonces: el Punto desde el que se continuará la exploración en la próxima iteración será el que ocupa el tope de la Pila p, E→↓↓, junto con el desplazamiento ↓ que se probará desde él en primer lugar, pues → ya se utilizó en la cuarta iteración; como se puede comprobar fácilmente, al desapilar el nuevo tope de la Pila o último Punto del Camino en construcción será ahora el Post-It (E→↓,←).
- **Octava Iteración de la Búsqueda de c:** recuperado por fin el puntoActual de la cuarta iteración, si se realiza el desplazamiento ↓ prescrito se alcanza un obstáculo, por lo que se intenta la posibilidad de desplazarse al Punto contiguo situado a su izquierda, que tiene éxito pues es un Punto libre, de donde puntoSiguiente = E→↓↓←. Así mismo, tras esta iteración se confirma que el último Punto del Camino en construcción es E→↓↓ y que desde él solo sería posible desplazarse hacia arriba, ↑, lo que se apunta en un Post-It que se apila en p, por si fuera necesario recordarlo en un momento posterior de la Búsqueda del Camino solución.
- . . .

  Ejercicio propuesto: continúese la traza ejemplo hasta obtener el Camino solución propuesto en el dibujo del Laberinto.

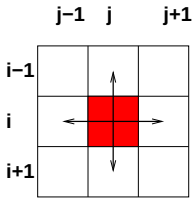
Alcanzado el punto de desarrollo actual del Esquema de Búsqueda de un Camino del Laberinto, para completarlo definitivamente basta con disponer de una Representación del Laberinto, esto es de un soporte en memoria de la Colección de Puntos que lo conforman; nótese que el soporte elegido determina la codificación de un Punto del Laberinto (sus coordenadas, acceso y si es o no un obstáculo), la talla del problema, la codificación de los tipos de desplazamiento permitidos y su ordenación y, finalmente pero no menos importante, la codificación de un Post-It de la Pila, que debe realizarse de forma congruente con la de Punto del Laberinto.

Así, si se escoge una Matriz int laberinto[][] de ceros y unos como soporte del Laberinto, cada una de sus componentes laberinto[i][j] representa un Punto p del Laberinto de coordenadas (i, j) tal que si laberinto[i][j] = 1 entonces p es obstáculo y sino es un Punto libre, laberinto[i][j] = 0. En lo que sigue, y sin que ello suponga pérdida de generalidad alguna, se supondrá que laberinto es una Matriz cuadrada (talla+2)×(talla+2) que representa un Laberinto de talla×talla Puntos y que desde cada uno de ellos solo es posible realizar cuatro tipos de desplazamientos: en este orden, derecha, debajo, izquierda y arriba. Nótese que la Matriz laberinto tiene 4∗talla+4 componentes más que Puntos el Laberinto para lograr un tratamiento uniforme de todas las componentes de la Matriz, tanto si representan bordes del Laberinto como Puntos interiores. Se rodea el Laberinto con un marco de obstáculos, esto es el valor de cada componente de las filas y columnas 0 y talla+1 de laberinto es uno. Por ejemplo, en la siguiente figura se muestra la Matriz 6×6 que representa un Laberinto

de 4×4 Puntos rodeado por un marco de obstáculos y cuya entrada y salida representan las componentes sombreadas:

1	1	1	1	1	1
1	0	0	0	0	1
1	1	0	1	0	1
1	1	0	0	1	1
1	1	1	0	0	1
1	1	1	1	1	1

De esta forma, como se muestra en la siguiente figura, desplazarse a la derecha o debajo o a la izquierda o arriba de un Punto cualquiera del Laberinto supone alcanzar desde su correspondiente componente de la Matriz, `laberinto[i][j]` tal que `1≤i≤talla AND 1≤j≤talla`, la componente `laberinto[i][j+1]` o `laberinto[i+1][j]` o `laberinto[i][j-1]` o `laberinto[i-1][j]` respectivamente, también de la Matriz `(talla+2)×(talla+2)`.



Ahora bien, para codificar y ordenar los tipos de desplazamiento permitidos en un Laberinto interesa observar que desplazarse desde la componente `laberinto[i][j]` no es más que sumar un Par de constantes a sus coordenadas; por ejemplo, para desplazarse a la derecha de `laberinto[i][j]`, a `laberinto[i][j+1]`, habrá sumarle la constante `0` a su fila `i` y la constante `1` a su columna `j`, de donde el Par `(0, 1)` representa un desplazamiento a la derecha de `laberinto[i][j]`; análogamente, el Par `(1, 0)` se asocia a debajo, el Par `(0, -1)` a izquierda y el Par `(-1, 0)` a arriba. Nótese que si cada uno de estos Pares es una componente de un `static final int[] SHIFT[]={{0, 1},{1, 0},{0, -1},{-1, 0}}` se logra codificar cada tipo de desplazamiento y su orden de aplicación mediante un único índice entero, el que indica su posición en el `array SHIFT`. En la siguiente tabla se resume este resultado:

Desplazamiento		
Tipo	Código o posición en `SHIFT`	Par a sumar o componente de `SHIFT`
derecha	0	`(0, 1)` ó `SHIFT[0]`
debajo	1	`(1, 0)` ó `SHIFT[1]`
izquierda	2	`(0, -1)` ó `SHIFT[2]`
arriba	3	`(-1, 0)` ó `SHIFT[3]`

En base a la descripción realizada de la Representación de un Laberinto, la estructura de una clase Java que lo represente podría tener la siguiente estructura (atributos y métodos):

```
package ejemplos.temaII_1;
import librerias.estructurasDeDatos.modelos.*;
import librerias.estructurasDeDatos.lineales.*;
public class Laberinto{
 protected int laberinto[][]; protected int talla;
 protected static final int [] SHIFT[] = {{0,1}, {1,0}, {0,-1}, {-1,0}};
 protected Pila<PostIt> p;
 public Laberinto(int talla){
     this.talla = talla;
     this.laberinto = new int[talla+2][talla+2];
     inicializarLaberinto();
     inicializarBordes();
 }
 private void inicializarLaberinto(){...}
 private void inicializarBordes(){...}
 public String toString(){...}
 public String buscarCamino(){
     String c = "Ninguno"; Pila<PostIt> p = new ArrayPila<PostIt>();
     /** Usando un Esquema de Búsqueda Anidada y una Pila p de Post-It se obtiene,
       * si existe, el camino c que une la entrada y la salida de this laberinto **/
     ...
     return c;
 }
}
```

donde `buscarCamino` transcribe a Java el Esquema de Búsqueda de `c` en el Laberinto y que no se podrá completar hasta no definir la clase `PostIt` que representa los Datos de la Pila. Dicha definición deberá ser congruente con la establecida para un Punto del Laberinto, pues como resultado de la Búsqueda del primer `puntoSiguiente` libre éste puede resultar ser tanto un Punto del Laberinto como el último Post-It de la Pila. Para tratar esta dicotomía de tipos de forma congruente, `PostIt` se debe definir como la siguiente clase **interna** de `Laberinto`:

```
class PostIt {
 /** Representación de las coordenadas del último Punto del Camino establecido,
   * laberinto[fila][columna]: **/
 int fila, columna;
 /** Representación del siguiente tipo de desplazamiento a realizar desde
   * laberinto[fila][columna], por si se vuelve a él al desandar un Camino **/
 int nextSHIFT;
 PostIt(int f, int c){ fila = f; columna = c; nextSHIFT = 0;}
 public String toString(){ return "("+fila+","+columna+")"; }
 public boolean equals(Object e){
     return ( fila == ((PostIt)e).fila && columna ==((PostIt)e).columna);
 }
 /** SII this.siguienteSHIFT < SHIFT.length */
 PostIt desplazar(){
     PostIt res = new PostIt(fila+SHIFT[nextSHIFT][0],columna+SHIFT[nextSHIFT][1]);
     nextSHIFT++;
     return res;
 }
}
```

Entonces, si para simplificar se considera que los Puntos libres de entrada y salida del Laberinto siempre se corresponden con, respectivamente, las componentes de valor cero `laberinto[1][1]` y `laberinto[talla][talla]`, `buscarCamino` se puede escribir como sigue:

```
public String buscarCamino(){
 String c = "Ninguno";
 PostIt entrada = new PostIt(1,1), salida = new PostIt(talla, talla);
 PostIt puntoActual = entrada; laberinto[1][1] = 1; boolean hayCamino = true;
 p = new ArrayPila<PostIt>();
 while ( hayCamino && !puntoActual.equals(salida) ){
     PostIt puntoSiguiente = null ; boolean siguienteLibre = false;
     while ( puntoActual.nextSHIFT < SHIFT.length && !siguienteLibre ) {
         puntoSiguiente = puntoActual.desplazar();
         if ( laberinto[puntoSiguiente.fila][puntoSiguiente.columna] == 0 )
             siguienteLibre = true;
     }
     if ( siguienteLibre ){
         p.apilar(puntoActual);
         laberinto[puntoSiguiente.fila][puntoSiguiente.columna] = 1;
         puntoActual = puntoSiguiente;
     }
     else if ( !p.esVacia() ) puntoActual = p.desapilar(); else hayCamino = false;
 }
 if ( puntoActual.equals(salida) ){ p.apilar(salida); c = p.toString(); }
 return c;
}
```

Ejercicios propuestos:

1. Dada la clase `Laberinto` antes presentada: coméntese con detalle su método `buscarCamino`; complétese el código del método auxiliar `inicializarBordes` de forma que al ejecutarlo se rodee con un marco de obstáculos el Laberinto de `talla`×`talla` Puntos generados aleatoriamente al `inicializarLaberinto`; finalmente, diséñese su método `toString()`.
2. Analícese el coste Temporal del método `buscarCamino` de la clase `Laberinto`.
3. Para manejar congruentemente los Puntos del Laberinto y los Post-It de una Pila se podría haber definido `laberinto` como una Matriz de `PostIt`; estúdiese la factibilidad y eficiencia de tal propuesta si se compara con la que se acaba de presentar.
4. En base a la estrategia de Búsqueda descrita en este apartado obténgase el Camino que une la entrada y salida del siguiente Laberinto e indíquese si es el más corto que existe.

0	0	0	0
1	0	1	0
1	0	0	1
1	1	0	0

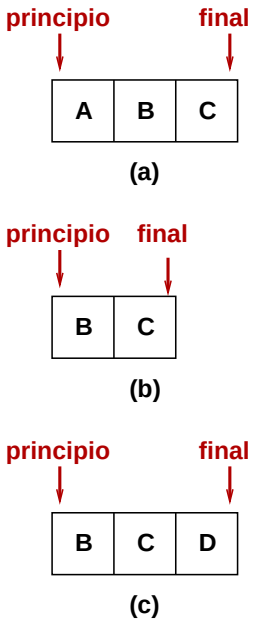
4. Cola

El significado de la palabra **Cola** resulta más que conocido, al igual que lo que se puede hacer con ella; a nadie se le escapa en qué situaciones se organiza una Cola, qué debe hacer cuando le indican que se ponga en Cola (operación `encolar`), qué lugar debe ocupar en ella para ser atendido (`primero()`) y qué debe hacer cuando terminen de atenderlo (`desencolar()`). Una EDA que plasme este tipo de gestión de los Datos se puede definir como sigue:

> una EDA **Cola** es una Colección Homogénea de Datos que se gestionan siguiendo un criterio **FIFO** (***F**irst **I**n **F**irst **O**ut*), esto es permitiendo únicamente el acceso al primero de sus Datos en orden de inserción.

Nótese entonces que, por definición de gestión FIFO, en la estructura de una Cola deben existir dos puntos distinguidos: su final para insertar y, en el extremo opuesto, su principio para recuperar y eliminar; además, y como ya ocurría con el tope de una Pila y el Punto de Interés de una Lista, ninguno de estos puntos aparece explícitamente como parámetro en las operaciones de Cola y su representación específica queda relegada a la fase de Implementación.

Con el siguiente ejemplo se ilustra gráficamente cómo se comporta la EDA Cola:



Inicialmente, figura (a), se dispone de una Cola de tres Datos de tipo `Character`, siendo `A` el `primero()` de ellos pues ocupa su principio; al `desencolar()`, figura (b), `A` se elimina y, aunque el principio de la Cola no se altera, el `primero()` de sus Datos es ahora `B`; finalmente, como muestra la figura (c), si ahora se inserta un nuevo Dato ejecutando «`encolar(new Character('D'))`» éste pasa a ser el último de la Cola, en la que el `primero()` sigue siendo `B`.

Ya para concluir, se deben reseñar dos cuestiones directamente relacionadas con el hecho de que el principio de una Cola es su único punto de acceso: si una Cola está vacía, lo que se puede comprobar con la operación `esVacia()`, la ejecución de las operaciones `primero()` y `desencolar()` provoca un error que se debe prevenir usándolas con la Precondición `!esVacia()`; `desencolar()` es la operación que permite acceder secuencialmente a todos los Datos de una Cola, pues elimina el Dato que ocupa su principio.

4.1. Modelo Java: la interfaz genérica `Cola`

El Modelo FIFO según el que se comporta una Cola se puede describir en Java mediante la siguiente interfaz genérica, donde el perfil del método `desencolar` aparece sobrecargado de la misma manera, y con el mismo fin, que el método `desapilar` de `Pila`:

```
package librerias.estructurasDeDatos.modelos;
/** Cola<E>: Base de la Jerarquía que representa el Modelo de una Cola de
 * Elementos de tipo E, o describe en Java las operaciones de una Cola **/
public interface Cola<E> {
// métodos Modificadores del estado de una Cola:
  /** inserta el Elemento e en el final de una Cola, o lo sitúa en su final **/
  void encolar(E e);
  /** SII !esVacia(): obtiene y elimina de una Cola el Elemento que ocupa su principio **/
  E desencolar();
// métodos Consultores del estado de una Cola:
  /** SII !esVacia(): obtiene el Elemento que ocupa el principio de una Cola,
    * el primero en orden de inserción **/
  E primero();
  /** comprueba si una Cola está vacía **/
  boolean esVacia();
}
```

4.2. Ejemplos de uso

Cualquiera de los siguientes enunciados describe un problema en cuya estrategia de resolución se usa una Cola; excepto el último, cuya solución se describe con detalle en esta sección, se propone **a modo de ejercicio** resolverlos:

1. Amplíese la Jerarquía de `Pila` con un método recursivo y dinámico `transformarEnCola` que convierta una Pila en una Cola cuyo último Dato sea el situado en el tope de la Pila. Además, suponiendo que los métodos de `Pila` y `Cola` tienen un coste Temporal independiente de la talla del problema, estímese el coste Espacial y el Temporal del método diseñado y discútase si un diseño iterativo los mejoraría.
2. Diséñese un método recursivo que invierta una Cola; discútase también su coste Espacial y Temporal y la conveniencia de implementarlo usando sólo los métodos de `Cola`.
3. Dada la clase `Laberinto` que se introdujo al resolver el problema del mismo nombre, añadirle un nuevo método `buscarCaminoMinimo` que obtenga, si existe, el Camino más corto que una los Puntos de entrada y salida de un Laberinto.

Nota: además de las interfaces `Pila` y `Cola` de `librerias.estructurasDeDatos.modelos`, cualquiera de los diseños propuestos debe disponer para poder ejecutarse del nombre de una Implementación de cada interfaz y del nombre del paquete en que éstas se ubican; para los diseños propuestos supónganse `ArrayPila` para `Pila` y `ArrayCola` para `Cola`, ambas clases en el paquete `librerias.estructurasDeDatos.lineales`.

El Problema del Camino Más Corto en un Laberinto

Utilizando una herramienta CAD (*Computer Aided Design*) se pueden diseñar automáticamente circuitos impresos e integrados en los que para minimizar el retraso de la señal es indispensable que el cable que una sus componentes entre sí discurra siguiendo el Camino Más Corto posible (*Wire Routing Problem*). Para lograrlo se dispone una malla sobre el área a cablear que la divide en una serie de Puntos que conforman una estructura parecida a la de un Laberinto y en la que los obstáculos son, precisamente para evitar cortocircuitos, aquellos Puntos por los que ya discurre el cable con el que se pretenden enlazar las componentes del circuito.

El problema que se acaba de enunciar es una de las instancias del Problema del Camino Más Corto en un Laberinto y su solución utiliza muchas de las ideas expuestas para resolver el del Laberinto; por ello se presenta en este punto como una modificación de la estrategia de Búsqueda de un Camino en un Laberinto (método `buscarCamino`), que se recuerda ahora:

> para encontrar, si existe, un Camino que une los Puntos de entrada `E` y salida `S` de un Laberinto es necesario buscar ordenadamente el siguiente Punto (`puntoSiguiente`) libre entre todos los que se puedan alcanzar desde el Punto actual de exploración (`puntoActual`); si en un momento dado `puntoActual` está rodeado de obstáculos habrá que retroceder hasta el último Punto del Camino construido -por ello con la ayuda de una Pila- y, si es posible, continuar la exploración desde él. La exploración termina bien cuando `puntoActual` coincide con `S` o bien cuando al volver hacia atrás no hay Punto del Camino alguno desde el que seguir explorando, i.e. no hay Camino en el Laberinto.

Si lo que se desea buscar no es un Camino cualquiera sino el más corto de todos los existentes la modificación esencial que habrá que realizar en tal estrategia será la forma en la que se establece el siguiente Punto de exploración a partir del actual:

> la Búsqueda del primer Punto libre alcanzable desde `puntoActual` pasa a ser ahora un Recorrido para obtener el primer Punto accesible a menor distancia de `puntoActual`.

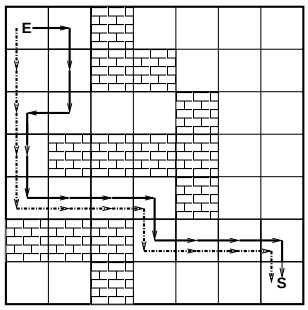
Obviamente, antes de establecer tal mínimo habrá que calcular la distancia entre `puntoActual` y sus adyacentes libres, esto es que **ni** sean obstáculos **ni** hayan actualizado aún su distancia en una iteración anterior desde otro `puntoActual` del que también son adyacentes; para ello, si `laberinto[puntoActual.fila][puntoActual.columna]` representa la distancia mínima acumulada desde `E` hasta `puntoActual` entonces,

- si `puntoActual == E`, `laberinto[puntoActual.fila][puntoActual.columna] = 0`, específicamente `laberinto[1][1] = 0`, es la distancia acumulada hasta la entrada `E`;
- si `puntoActual != E`, `laberinto[puntoActual.fila][puntoActual.columna]+1` es la distancia acumulada desde `E` hasta un Punto libre adyacente a `puntoActual`;

Con estas definiciones es posible calcular sin ayuda de una Matriz auxiliar, sobre la propia Matriz `laberinto`, las distancias mínimas desde `E` hasta cada Punto accesible del Laberinto en cada iteración de la Búsqueda del Camino Más Corto: los adyacentes libres alcanzables desde `E` tienen una distancia acumulada igual a `1`, los vecinos libres de éstos igual a `2` y así sucesivamente hasta alcanzar `S` o bien hasta que no quede vecino alguno por explorar. Nótese que esta actualización de distancias únicamente se puede realizar **antes** de marcar como visitado

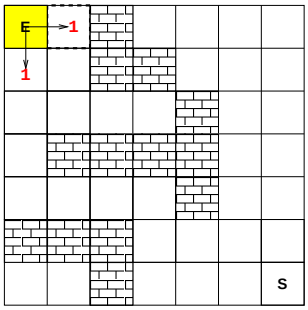
`puntoActual`, lo que no supone cambio semántico alguno con respecto a la estrategia del Problema del Laberinto: en lugar de cada `puntoSiguiente` encontrado, basta marcar como visitado `puntoActual` antes de actualizar su contenido al final de cada iteración.

En principio, modificando tan solo y como se ha indicado la determinación del siguiente Punto de exploración ya se dispone de una estrategia de resolución del Problema del Camino Más Corto. Ahora bien, antes de trasladarla a código Java resulta razonable aplicarla a un Laberinto conocido y, en su caso, refinarla; sirva como ejemplo el que se utilizó para ilustrar la solución del Problema del Laberinto, del que la siguiente figura muestra su representación gráfica, el Camino resultado del método `buscarCamino` (de longitud `14`, en línea continua y gruesa) y uno de los dos Caminos Más Cortos (de longitud `12`) entre su entrada y su salida:



Para construir este Camino Más Corto del Laberinto ejemplo, que representará `cMinimo` en adelante, los pasos que dicta la estrategia hasta ahora propuesta son:

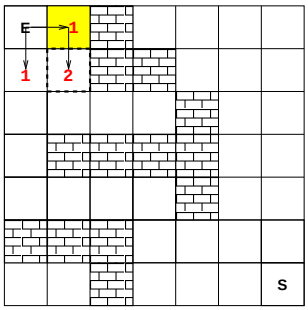
- **Inicialización de la Búsqueda de `cMinimo`:** se supone que existe un Camino y que su primer Punto es `E`. Por tanto, inicialmente `hayCamino = true`, se crea la Pila auxiliar `p` de `PostIt` vacía y `puntoActual = E` con distancia acumulada `laberinto[1][1] = 0`.
- **Primera iteración de la Búsqueda de `cMinimo`:** desde `puntoActual = E` se actualizan las distancias a sus adyacentes libres, por lo que en este orden `laberinto[1][2]=1` y `laberinto[2][1]=1`. Como `E` no se puede volver a visitar `...` ¿cuál de ellos es el mínimo, el `puntoSiguiente` a explorar en la próxima iteración? Como muestra la siguiente figura, donde `puntoActual` aparece sombreado y `puntoSiguiente` enmarcado por una línea discontinua, si todos los Puntos accesibles desde `puntoActual` son equidistantes se considera que `puntoSiguiente` es el alcanzado en **primer** lugar según el orden establecido para los desplazamientos en el Laberinto; en el ejemplo `laberinto[1][2]` es el **primer** mínimo.



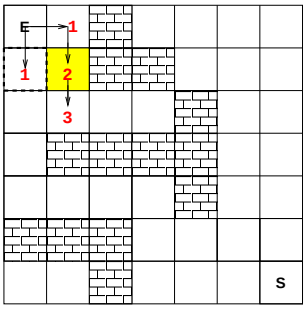
Antes de concluir la iteración, se apila en `p` el `PostIt` que contiene las coordenadas de `E` y cuyo siguiente desplazamiento posible es `nextSHIFT = 1`, i.e. el siguiente al que ha permitido alcanzar el primer mínimo desde `E`.

Breve disgresión: el lector atento puede haber descubierto ya un pequeño problema en la estrategia propuesta: el valor de la Matriz `laberinto` para un obstáculo, `1`, coincide con el de la distancia de `E` a sus adyacentes libres; aunque la representación gráfica del Laberinto lo obvia, su codificación no puede hacer lo mismo. La solución a este problema es fácil: mantener el valor actual `1` para obstáculos y Puntos visitados e inicializar a dos la distancia a `E`, `laberinto[1][1]=2`; así, cuando `puntoActual` sea `S` la distancia real del Camino Más Corto será la acumulada en `S` menos dos, i.e. `laberinto[talla][talla]-2`. Este cambio no se aplicará a la traza en curso.

- **Segunda iteración de la Búsqueda de `cMinimo`:** desde `puntoActual=E`→ el único Punto libre alcanzable es `laberinto[2][2]`, el `puntoSiguiente` desde el que comenzará la exploración en la siguiente iteración; la siguiente imagen muestra que sin embargo el Punto más cercano a `E` es `laberinto[2][1]`, lo que supone una ligera desviación en el cálculo de `cMinimo`. Aún así `puntoActual` se marca como visitado y se apila en `p` con `nextShift=2`.



- **Tercera iteración de la Búsqueda de `cMinimo`:** aunque el único adyacente libre a `puntoActual = E`→↓ es `laberinto[3][2]`, el Punto accesible a menor distancia de `puntoActual` ¡ y también de `E` ! es `laberinto[2][1]`.



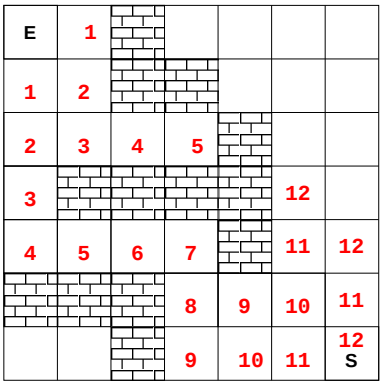
Ahora bien, que `laberinto[2][1]` sea el siguiente Punto de exploración no termina de corregir la desviación en la construcción de `cMinimo` ya señalada: `puntoActual` se apilará en `p` antes de concluir la iteración y en ella permanecerá hasta el final de la Búsqueda aunque no forme parte del Camino Más Corto, exactamente igual que el `puntoActual` de la segunda iteración `E`→. Aunque esto no invalida la estrategia sí lleva a plantear una cuestión: si al alcanzar la salida del Laberinto la Pila ya no contiene los Puntos de `cMinimo`, **¿cómo y de dónde obtener el Camino Mínimo con esta estrategia?**
Directamente relacionada con la elección de `laberinto[2][1]` surge otra cuestión no menos importante: si ya desde la primera iteración se sabe positivamente que es equidistante a `laberinto[1][2]`, **¿por qué esperar a explorarlo en la cuarta?**

Las dos cuestiones que se acaban de realizar sugieren detener la traza y replantear la estrategia seguida hasta el momento. Empezando por la última cuestión, la idea clave a aprehender es que **no** hace falta calcular explícitamente el mínimo accesible desde `puntoActual` si ya se sabe que los primeros Puntos alcanzados desde él son también los más cercanos y, por tanto, los primeros a explorar, independientemente de si hay más de uno equidistante. Pero entonces es una Cola `q` y no una Pila la que permite recordar el orden en el que se van alcanzando los adyacentes libres a `puntoActual` y realizar su gestión FIFO «siguiente alcanzado, siguiente a explorar» , en lugar de la LIFO actual «último alcanzado, primero a explorar» que obliga a calcular el vecino accesible a distancia mínima de `puntoActual`. En efecto, si inicialmente `puntoActual = E` y `q` está vacía, en cada iteración de la Búsqueda de `cMinimo` basta ir encolando en `q` los adyacentes libres a `puntoActual` conforme son alcanzandos y se actualizan sus distancias; una vez hecho esto y cara a la siguiente iteración, «`if ( !q.esVacia() ) puntoActual = q.desencolar();`». Así hasta que el bucle termine porque `puntoActual` es `S` o porque `q.esVacia()` y no hay Camino que una los Puntos de entrada y salida del Laberinto (`hayCamino = false`). El siguiente código Java resume lo dicho sobre la Cola `q` y su papel en la estrategia de Búsqueda del Camino Más Corto en un Laberinto; como se puede observar no figura en él la resolución de la Búsqueda u obtención de `cMinimo` tras alcanzar `S`, pues esta cuestión aún está pendiente de ser resuelta.

```
public String buscarCaminoMinimo(){
 String cMinimo = "Ninguno";
 PostIt entrada = new PostIt(1,1), salida = new PostIt(talla, talla);
 PostIt puntoActual = entrada; laberinto[1][1] = 2; boolean hayCamino = true;
 q = new ArrayCola<PostIt>();
 while ( hayCamino && !puntoActual.equals(salida) ){
     distanciaAdyacente = laberinto[puntoActual.fila][puntoActual.columna] + 1;
     while ( puntoActual.siguienteSHIFT < SHIFT.length ) {
         puntoSiguiente = puntoActual.desplazar();
         if ( laberinto[puntoSiguiente.fila][puntoSiguiente.columna] == 0 ){
            laberinto[puntoSiguiente.fila][puntoSiguiente.columna] = distanciaAdyacente;
            q.encolar(puntoSiguiente);
         }
     }
    if ( !q.esVacia() ) puntoActual = q.desencolar();
    else hayCamino = false;
 }
 if ( puntoActual.equals(salida) ){...}
 return cMinimo;
}
```

Nótese que el coste Temporal de `buscarCaminoMinimo` es el del bucle de Búsqueda propiamente dicho, i.e. el de la exploración de todos aquellos Puntos del Laberinto que son susceptibles de pertenecer al Camino `cMinimo` en construcción; como en el Peor de los Casos cualquier adyacente libre a uno dado puede ser el siguiente `puntoActual`, cualquier Punto del Laberinto es susceptible de serlo y el bucle tardará en ejecutarse un tiempo proporcional a $\texttt{talla}^2$. Aunque esta cota no se puede mejorar, `laberinto` siempre es una Matriz cuadrada de $\texttt{talla}^2$ componentes, cabe resaltar que el uso de una Cola es más efectivo que el de una Pila: evita la implementación del cálculo del vecino accesible a distancia mínima de `puntoActual` y, adicionalmente, no requiere marcar como visitado Punto del Laberinto alguno pues con desencolarlo es suficiente.

Ya para terminar se analiza la obtención de `cMinimo` al concluir el bucle de Búsqueda con `puntoActual = S`. Para ello, suponiendo `laberinto[1][1] = 0`, obsérvese el estado del Laberinto ejemplo en ese momento -quedando propuesto trazar su ejecución para comprobarlo:



La Matriz `laberinto` se ha transformado tras la ejecución del bucle en la Matriz de distancias mínimas desde `E` a cada Punto del Laberinto, por lo que `laberinto[talla][talla]` es el valor de la longitud de `cMinimo`. Para obtener los Puntos que lo conforman basta, empezando en `puntoActual = S`, con acceder siempre a aquel Punto que tiene una longitud (valor de `laberinto`) una unidad menor que la actual. Por tanto, la resolución de la Búsqueda de `cMinimo` es un Recorrido en el que se visitan desde `S` hasta `E` tantos Puntos del Laberinto como longitud tenga `cMinimo`, lineal con `talla`; nótese que entonces $\texttt{talla}^2$, el coste del cálculo de la Matriz de distancias, sigue estableciendo la cota superior del problema.

Ejercicios propuestos:

1. Complétese el diseño del método `buscarCaminoMinimo` para que obtenga `cMinimo` a partir de la Matriz de distancias `laberinto`.

2. En la siguiente figura se muestra un Laberinto con entrada en el Punto `(3, 2)` y salida en el `(4, 6)`; realícense dos trazas gráficas sobre él, una con estrategia de Búsqueda del Camino Más Corto que usa una Pila y otra con la que usa una Cola.