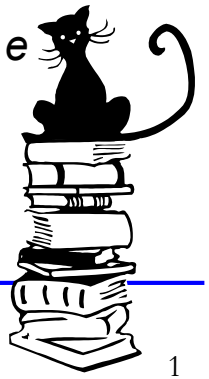
# Tema II: Tipos de Datos Lineales

*Secuencias de elementos*
*de un cierto tipo*
*dispuestos en una dimensión*

# Bibliografía


- Capítulo 3 del libro "Weiss, M.A.: *Data Structures and Algorithm Analysis in C++, Fourth Edition*, Pearson/Addison Wesley, 2014."
- *Capítulo 4 del libro "Clifford A. Shaffer: Data Structures and Algorithm Analysis, Edition 3.2 (C++ Version) March 2013."*
- *Capítulo 4 del libro "Z.J. Hernández y otros: Fundamentos de Estructuras de Datos. Soluciones en Ada, Java y C++, Thomson, 2005."*
- Tema II del libro de apuntes "Campos Laclaustra, J.: Apuntes de *Estructuras de Datos y Algoritmos*, *segunda edición*, 2018 (versión 4, 2022).
- *Capítulos 6, 7, 4 y 5 del libro "Joyanes, L., Zahonero, I., Fernández, M. y Sánchez, L.: Estructuras de datos. Libro de problemas, McGraw Hill, 1999.",*
- *Capítulos 3, 4 y 5 del libro "*Martí Oliet, N., Ortega Mallén, Y., Verdejo López, J.A.: *Estructuras de datos y métodos algorítmicos. 2ª edición: 213 ejercicios resueltos*. Garceta, 2013*",*
- *Capítulo 3 del libro "X. Franch: Estructuras de datos. Especificación, diseño e implementación, 3ª edición, Ediciones UPC, 2001."*
- *Y otros....*

# Tipos lineales

- Permiten almacenar relaciones de orden o secuencia entre los elementos:

  *Secuencias de elementos de un cierto tipo dispuestos en una dimensión*

  ***e1 e2 e3 e4 e5 e6 e7 … eN***

  ***A C X B A D P … E***

*Toda secuencia refleja un orden en sus elementos, resultado de colocarlos en una dimensión*

  - Mucha variedad de posibles TADs:
    - Con o sin elementos repetidos
    - Criterios de orden en la secuencia:
      - » respecto a los propios datos en los elementos (Ej.: por orden alfabético),
      - » respecto a criterios externos a los datos (Ej.: por orden de llegada o inserción)
    - Operaciones típicas:
      - » *crear, añadir, esVacía?, quitar, pertenece, tamaño, …*
    - Pueden ser operaciones:
      - » guiadas por el orden de los datos o de la secuencia
      - » limitadas a determinados extremos de la secuencia
      - » relativas al último elemento accedido o insertado
      - » relativas a la posición en la secuencia
    - Operaciones de acceso y recorrido (iteradores) → según el orden en la secuencia
      - » primero, siguiente, esÚltimo?, último, esPrimero?, anterior, …

# Listas.
# TAD lista con acceso por los extremos.
# Implementación estática.

Lección 6

# Esquema

→ TAD Lista con acceso por los extremos (bicola)

- Operaciones de acceso y manipulación limitadas a ambos extremos de la secuencia
- Implementación en memoria estática

– listas con operaciones de recorrido (iterador)

*Los valores del TAD lista representan secuencias de 0 o más elementos, con operaciones de acceso y manipulación en ambos extremos de la secuencia. …*

- Caso especial: lista vacía

- Caso general: lista no vacía

# Listas genéricas con acceso por ambos extremos. Especificación.

**espec** listasGenéricas
**usa** boleanos,naturales
**parámetro formal**
**género** elemento
**fpf**
**género** lista
*{Los valores del TAD lista representan secuencias de 0 o más elementos, con operaciones de acceso y manipulación en ambos extremos de la secuencia. Para recorrer los elementos de la secuencia, ofrece las operaciones de un Iterador, definido sobre las listas en sentido de primero a último}*
**operaciones**
crear: →lista
*{ Devuelve una lista vacía, sin elementos}*
añadirPrimero: elemento e, lista l → lista
*{Devuelve la lista igual a la resultante de añadir e como primer elemento en l}*
añadirÚltimo: lista l, elemento e → lista
*{ Devuelve la lista igual a la resultante de añadir e como último elemento en l}*
esVacía?: lista l → booleano
*{ Devuelve verdad si y sólo si l no tiene elementos}*
**…**

# Listas genéricas con acceso por ambos extremos. Especificación.

**parcial** borrarÚltimo: lista l → lista
*{ Devuelve la lista igual a la resultante de borrar el último elemento de l;*
*Parcial: la operación no está definida si l es vacía }*
**parcial** borrarPrimero: lista l → lista
*{Devuelve la lista igual a la resultante de borrar el primer elemento de l;*
*Parcial: la operación no está definida si l es vacía }*

**parcial** primero: lista l → elemento
*{Devuelve el primer elemento de l;*
*Parcial: la operación no está definida si l es vacía }*
**parcial** último: lista l → elemento
*{Devuelve el último elemento de l;*
*Parcial: la operación no está definida si l es vacía }*

longitud: lista l → natural
*{Devuelve el número de elementos de l}*

*… {al final añadiremos las operaciones de un iterador}*

**Nota:** podría hacerse una especificación distinta, de forma que borrar (último o primero) de una lista vacía devuelva una lista vacía, y no sean parciales

**primero y último tienen que ser operaciones parciales**

**fespec**

# Implementación en memoria estática

- En pseudocódigo:

***LA DOCUMENTACIÓN SE ESCRIBE DESDE EL PRINCIPIO, NO SE DEJA TODA ELLA PARA EL FINAL***

**módulo genérico** listasGenéricas
**parámetros**
    **tipo** elemento
**exporta**
    **tipo** lista
    **procedimiento** crear **(sal** l:lista**)**
        … *{y cabeceras del resto de las operaciones de la especificación}*
**implementación**
    *{ Representación interna, e implementación de las operaciones}*
    …
**fin**

***{Con la documentación pública para los programadores usuarios de este módulo (sin detalles sobre la implementación interna) }***

***{Con la documentación privada para los programadores desarrolladores (actuales o futuros) de este módulo:***
- ***debe corresponderse con la descripción pública***
- ***con detalles sobre la implementación interna }***

# Implementación en memoria estática

- Representación interna en pseudocódigo:

```
constante max= …;
Tipo lista = registro
        datos: vector [1..max] de elemento;
        último: 0..max;
    Freg
```

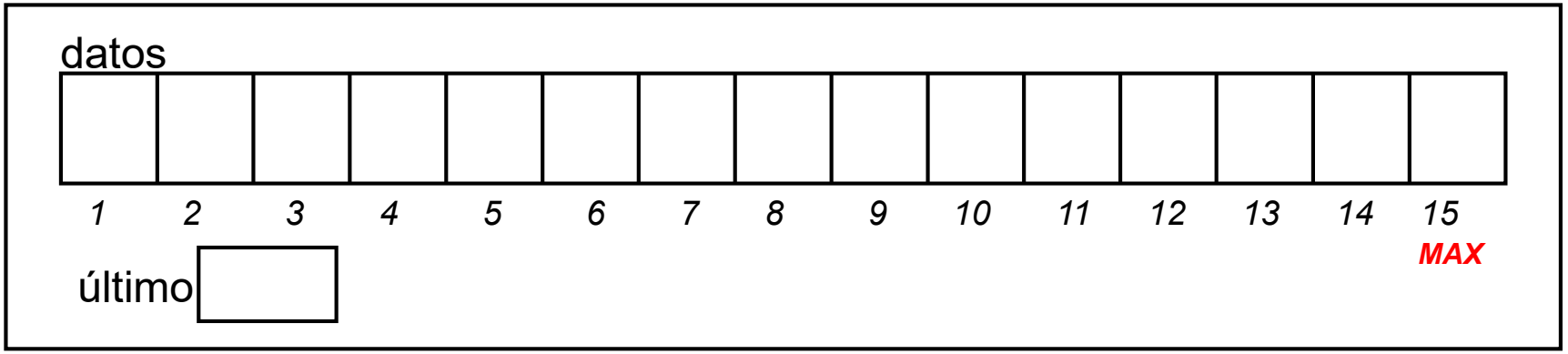
*Θ(Max) en memoria*

**Observaciones respecto a los vectores (arrays):**

- **Al declarar una variable de un tipo vector se reserva una zona contigua de memoria para poder almacenar datos en todas las componentes del vector**
- **Una vez creado, el vector no cambia de tamaño**

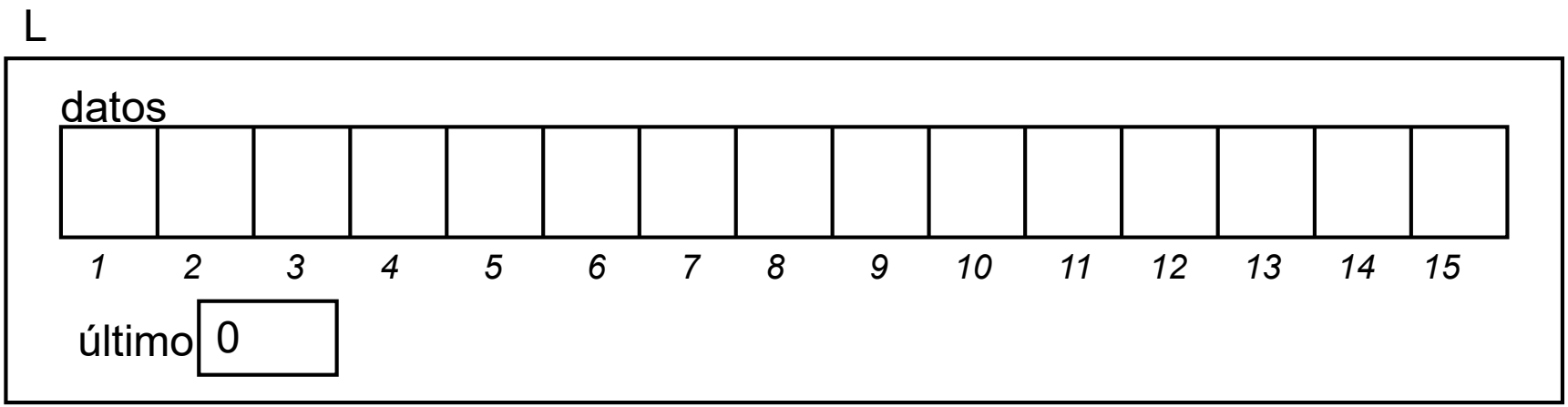
***{DOCUMENTACIÓN:*** *Los datos de la secuencia (o lista) se guardarán en un vector de elementos, de tamaño* ***max****, del cual usaremos las primeras posiciones (sin dejar huecos). Los datos se guardarán en el mismo orden en el que formen parte de la secuencia, por tanto el primer elemento (si existe) estará siempre en la componente 1 del vector. El campo último tendrá el número de posiciones ocupadas del vector con datos de elementos pertenecientes a la secuencia (lista), y tendrá valor 0 cuando la secuencia sea vacía. }* → Limitación del tamaño de la lista (secuencia), **max** elementos, que no existía en la especificación, y debe indicarse en la documentación de la Interfaz del módulo

lista



# Implementación estática

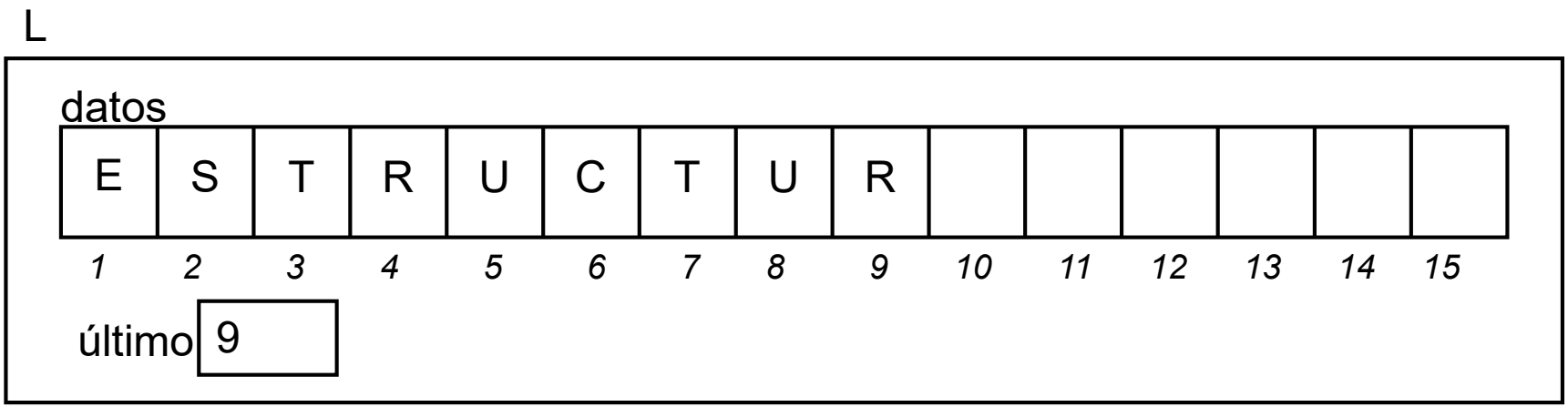
- **Implementación de las operaciones:**




**procedimiento** crear(**sal** L:lista)
*{**Post:** devuelve una lista vacía, sin elementos}*
**principio**
    L.último:=0;
**fin**

*Θ(1) en tiempo*

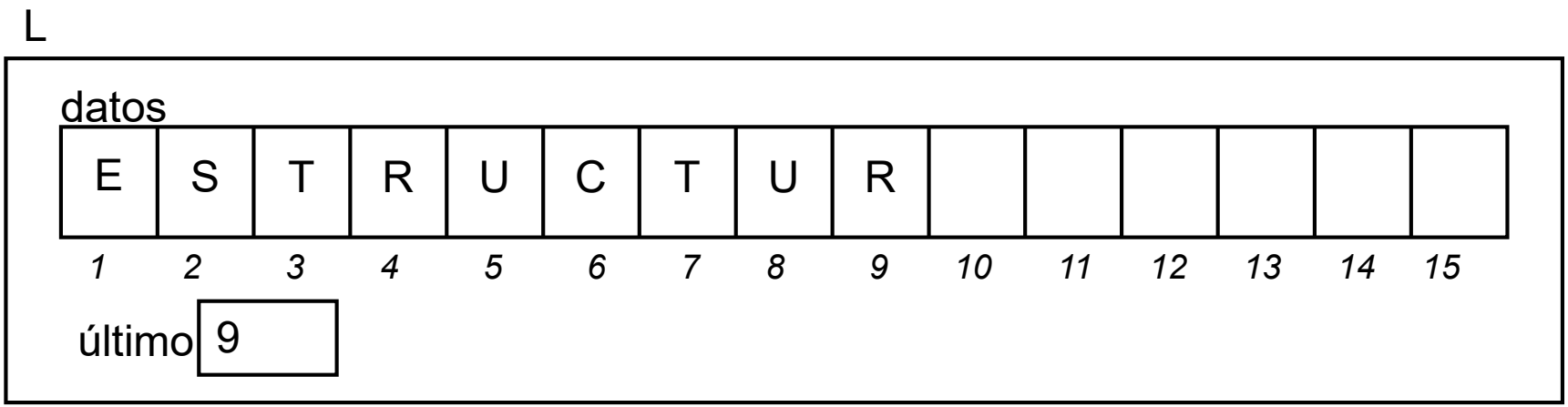
# Implementación estática




**función** esVacía?(L:lista) **devuelve** booleano
*{**Post:** devuelve verdad si y sólo si L no tiene elementos}*
**principio**
    **devuelve** (L.último=0)
**fin**

*Θ(1) en tiempo*

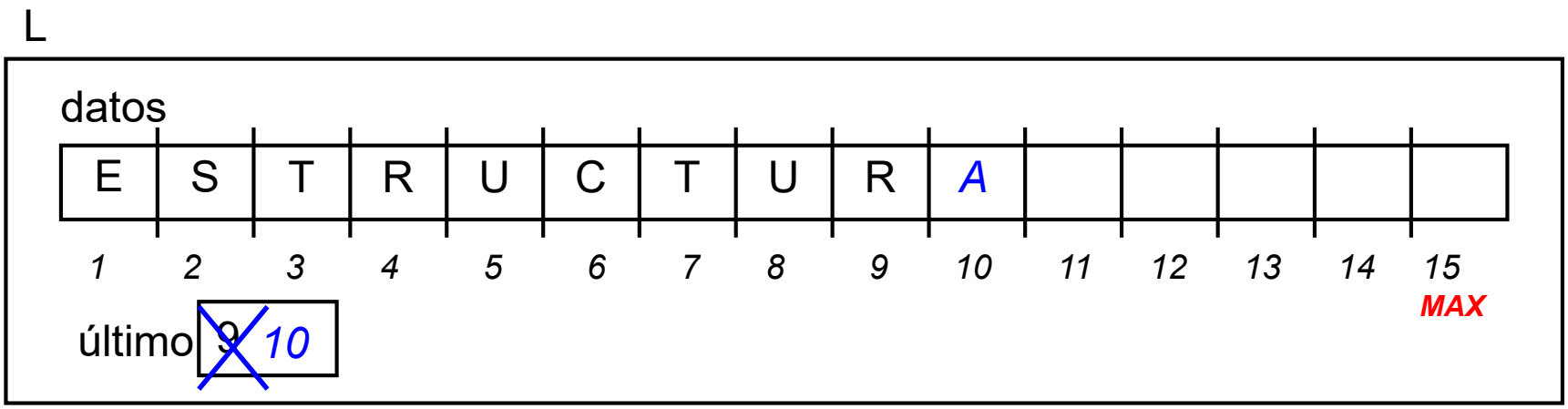
# Implementación estática




**función** longitud(L:lista) **devuelve** natural
*{**Post:** devuelve el número de elementos de L}*
**principio**
    **devuelve** (L.último)
**fin**

*Θ(1) en tiempo*

# Implementación estática




**procedimiento** añadirÚltimo(**e/s** L:lista; **ent** e:elemento; **sal** error: booleano)

***{Post: devuelve la lista L con e añadido como último elemento de L y error=falso, si es posible añadir el elemento; en caso contrario, L queda igual y devuelve error=verdad}***

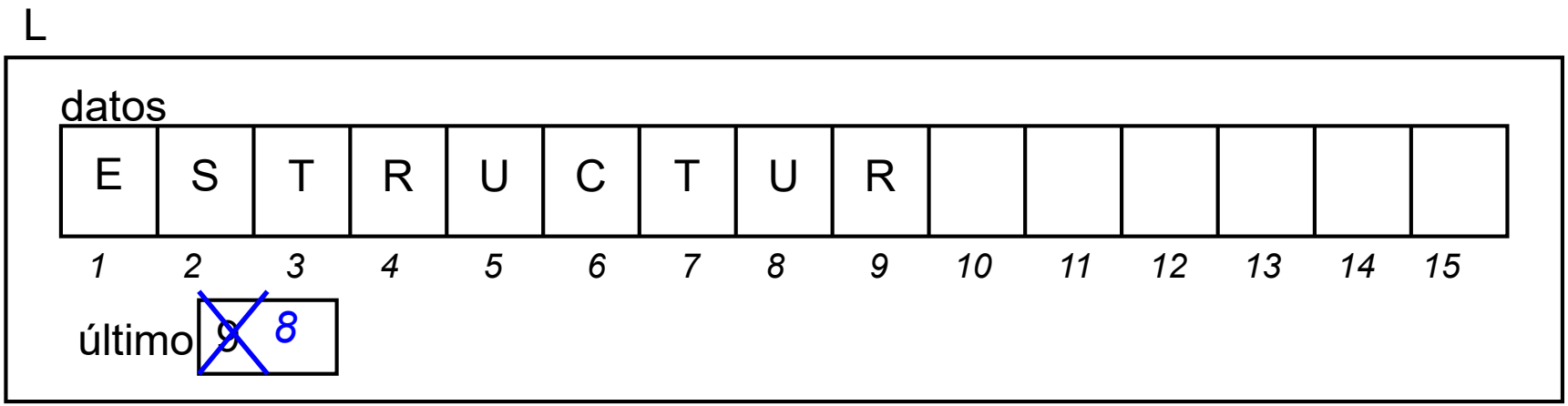
```
principio
    error:= falso;
     si (L.último<max) entonces
            L.último:=L.último+1;
            L.datos[L.ultimo]:=e
    sino
            error:=verdad;
    fsi
fin
```

*Θ(1) en tiempo*

*Operación implementada como parcial debido a la representación interna elegida para la implementación:*

*→ capacidad limitada del vector*

# Implementación estática




**procedimiento** borrarÚltimo(**e/s** L:lista ; **sal** error: booleano)

*{**Post:** si L es no vacía, devuelve la lista tras borrar el último elemento de L, y error falso; si L es vacía queda igual, y se devuelve error con valor verdad (operación parcial) }*

**principio**

```
    error:= (L.último=0);
     si (L.último>0) entonces
            L.último:=L.último-1
    fsi
```

**fin**

*Θ(1) en tiempo*

*Operación parcial: no se puede borrar el último elemento si la lista está vacía*

**parcial** último: lista l → elemento
*{Devuelve el último elemento de l; Parcial: la operación no está definida si l vacía }*
**parcial** primero: lista l → elemento
*{Devuelve el primer elemento de l; Parcial: la operación no está definida si l vacía }*

- **Operaciones *último* y *primero*:** *Θ(1) en tiempo*
  - implementación trivial
  - operaciones parciales → *no se puede devolver un elemento si la lista está vacía*
- **Opciones si no se cuenta con mecanismo programación con excepciones:**

  **a)** **función** último(L:lista) **devuelve** elemento
  *{* ***Pre:*** *L no es lista vacía.*
  ***Post:*** *si L es no vacía, devuelve el último elemento de L }*

  ***→ Implementación no robusta a menos que se programe con excepciones***

  **b)** **procedimiento** último(**ent** L:lista; **sal** e: elemento; **sal** error: booleano)
  *{****Post:*** *si L es no vacía, devuelve el último elemento de L, y error con valor falso; si L es vacía, se devuelve error con valor verdad (operación parcial) }*

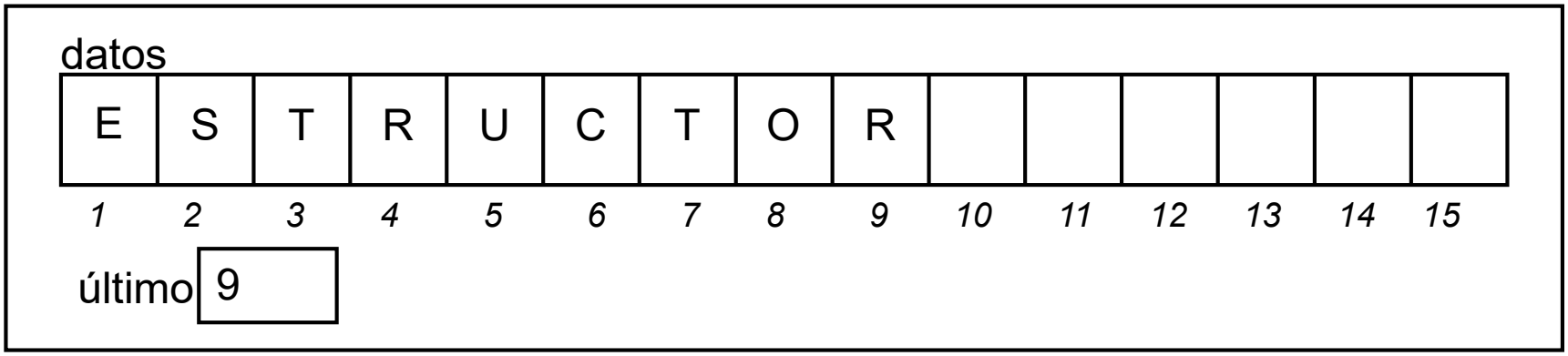
  **c)** **función** último(L:lista) **devuelve** elemento
  *{****Post:*** *si L es no vacía, devuelve el último elemento de L; si L es vacía, se produce un error y devuelve un elemento con valor especial que reservamos como no válido, y que no deberá poder añadirse nunca a la lista }*
  *→ PROBLEMA: Opción que restringe el uso del TAD, y difícilmente aplicable en implementaciones genéricas*

**procedimiento** añadirPrimero(**ent** e:elemento; **e/s** L:lista ; **sal** error: booleano)

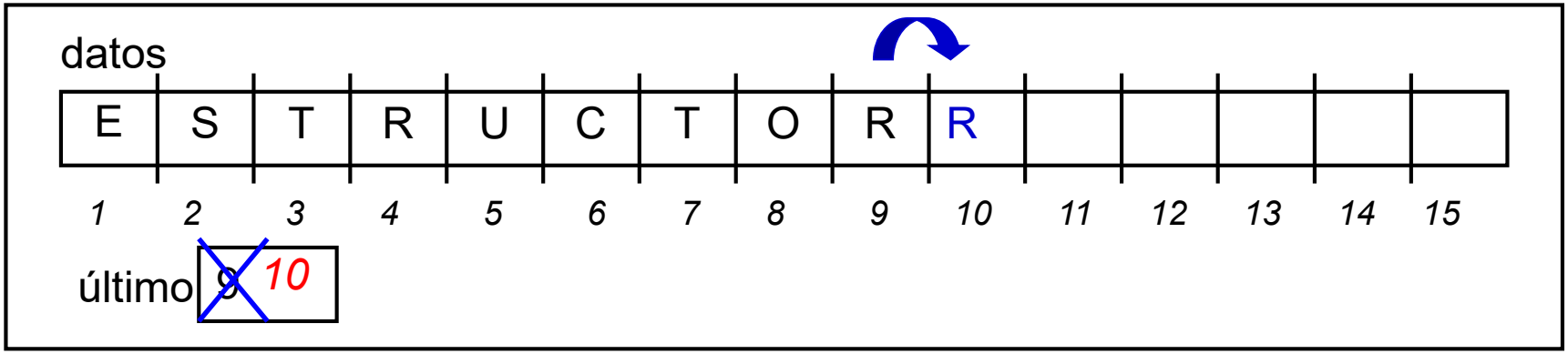
*{**Post:** devuelve la lista tras añadir e como primer elemento de L y error=falso, si es posible añadir el elemento; en caso contrario, L queda igual y devuelve error=verdad (operación implementada como parcial debido a la representación interna elegida para la implementación: capacidad limitada del vector) }*

*añadirPrimero(**'D'**, L, er)*

L

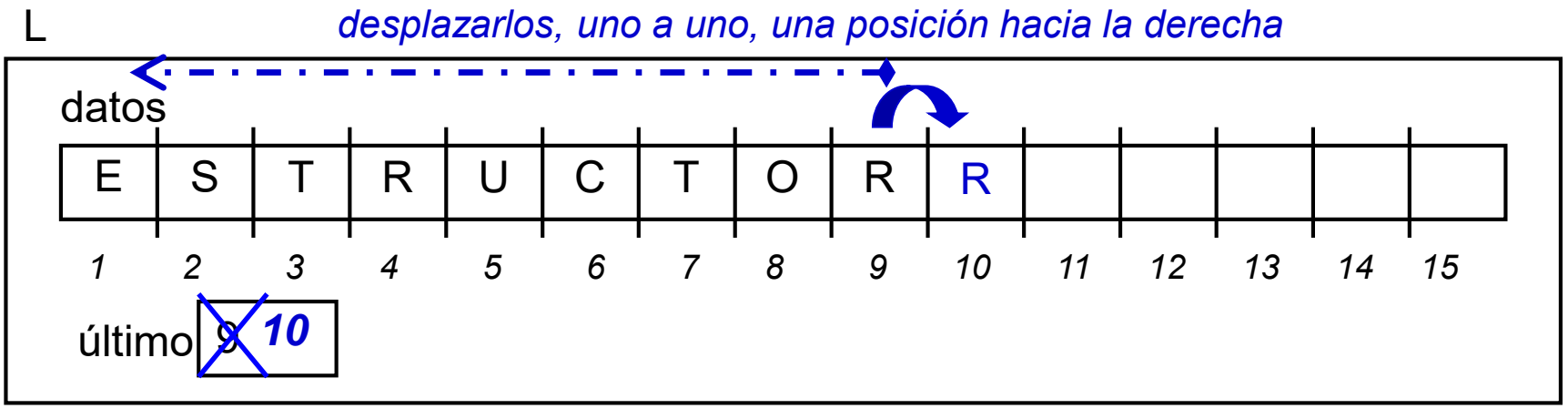


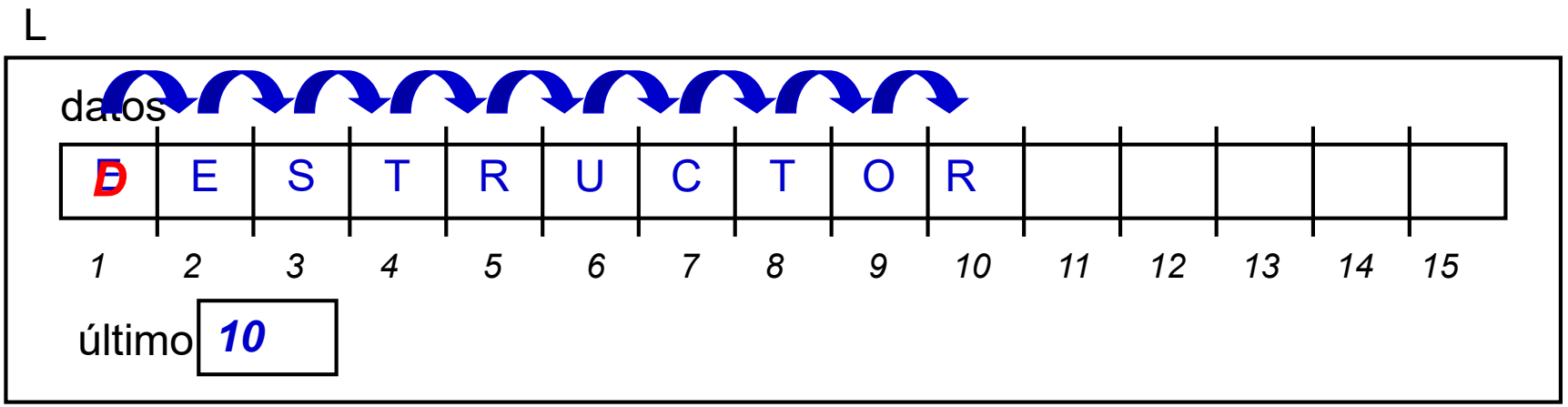

L



**procedimiento** añadirPrimero(**ent** e:elemento; **e/s** L:lista ; **sal** error: booleano)

*{**Post:** devuelve la lista tras añadir e como primer elemento de L…..}*



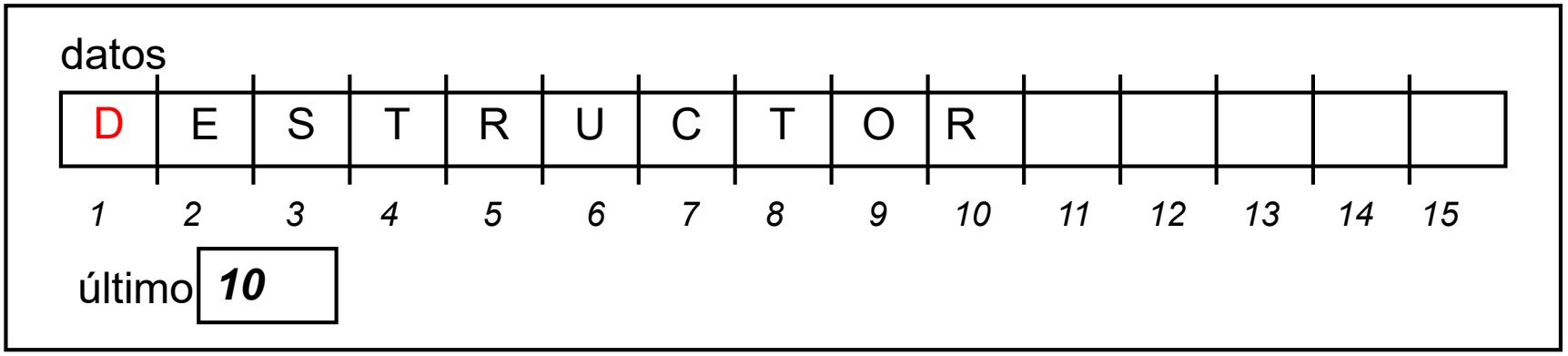




*Θ(N) en tiempo*

**procedimiento** borrarPrimero(**e/s** L:lista; **sal** error: booleano)

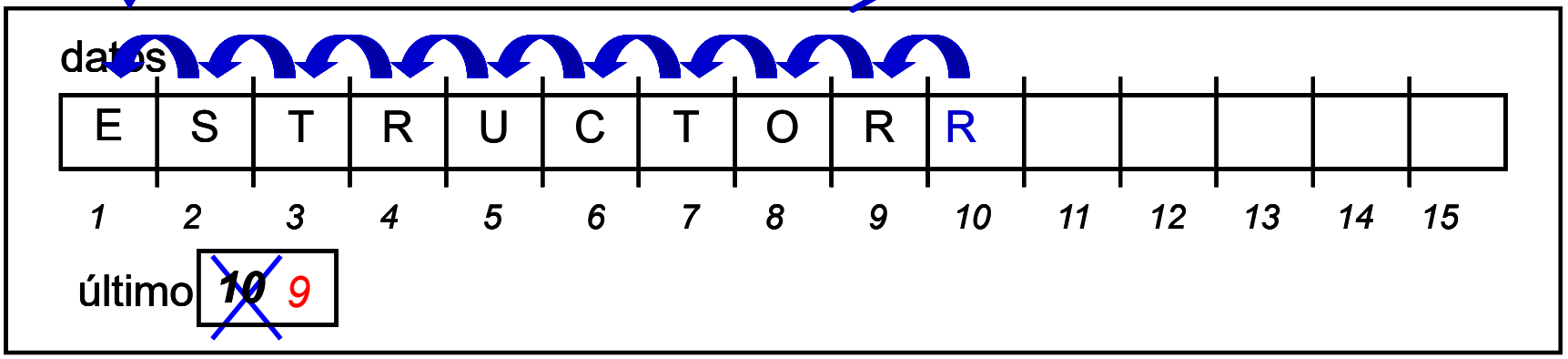
*{**Post:** si L es no vacía, devuelve la lista tras borrar el primer elemento de L, y en error falso; si L es vacía queda igual, y en error devuelve verdad (operación parcial) }*

L




*desplazarlos, uno a uno, una posición hacia la izquierda*

L




*Θ(N) en tiempo*

# Implementación estática

- Inconvenientes:
  - Coste espacio en memoria para el máximo (*max*) de elementos previstos
    - ¿Capacidad máxima?
    - Memoria reservada no utilizada
  - Introduce una limitación en el tamaño de la lista (secuencia) que no estaba en la especificación
    - Operaciones *añadirPrimero* y *añadirÚltimo* deben implementarse como operaciones parciales debido a la capacidad limitada del vector
  - Todas las operaciones $\Theta(1)$ en tiempo, salvo las operaciones *añadirPrimero y borrarPrimero* que tienen $\Theta(N)$ en tiempo:
    - desplazarán elementos para evitar dejar "huecos" (costes lineales)
    - Costes inadmisibles para grandes dimensiones y/o operaciones muy frecuentes
  - Se necesita **implementar** operaciones de ***igualdad*** y ***asignación*** entre listas

# Implementación estática

*{ Si hay que añadir esta operación al TAD (a la especificación, y luego a la implementación) el tipo elemento tendrá una restricción: tener definida la operación de comparación por igualdad (=) }*

- Operación de comparación de igualdad:

```
función iguales (L1,L2:lista) devuelve booleano
{Post: devuelve verdad si y solo si la lista L2 contiene los mismos
    elementos y en el mismo orden que la lista L1}
  variables
    iguales: booleano;
    i: natural:=1;
principio
    iguales:=(L2.último=L1.último);
    mientrasQue ((i<=L1.último) and iguales) hacer
          iguales:= (L1.datos[i]=L2.datos[i]);
           i:=i+1;
    fmq;
    devuelve iguales
fin
```

*No deben compararse las posiciones del vector "no ocupadas" o se producirán falsos negativos*

$\Theta(N)$ en tiempo en caso peor

# Implementación estática

- Operación de duplicación o copia:

```
procedimiento copiar(ent L1:lista; sal L2:lista)
{Post: devuelve la lista L2 resultante de copiar en ella todos los
    elementos de L1, en el mismo orden}
    variables i: natural:=1;
principio
    L2.último:=L1.último;
    mientrasQue (i<=L1.último) hacer
            L2.datos[i]:=L1.datos[i];
            i:=i+1;
    fmq;
fin
```

*Eficiencia → No copiar las posiciones del vector "no ocupadas"*

*Θ(N) en tiempo*

# Especificación de recorridos en listas genéricas

**espec** listasGenéricas

**…** *{… Para recorrer los elementos de la secuencia, ofrece las operaciones de un Iterador, definido sobre las listas en sentido de primero a último}*

**operaciones**

**…**

iniciarIterador: lista l → lista

*{ Prepara el iterador y su cursor para que el siguiente elemento a visitar sea el primero de la lista l (situación de no haber visitado ningún elemento)}*

existeSiguiente?: lista l →booleano

*{ Devuelve falso si ya se ha visitado el último elemento, devuelve verdad en caso contrario}*

***En cualquiera de los TADs contenedores que veamos será posible tener operaciones como estas para ofrecer un iterador***

**parcial** siguiente: lista l →elemento

*{ Devuelve el siguiente elemento de l.*

*Parcial: la operación no está definida si no existeSiguiente?(l) }*

**parcial** avanza: lista l → lista

*{ Devuelve la lista resultante de avanzar el cursor en l.*

*Parcial: la operación no está definida si no existeSiguiente?(l) }*

***Normalmente al implementar serán una única operación, que equivaldría a utilizar:***

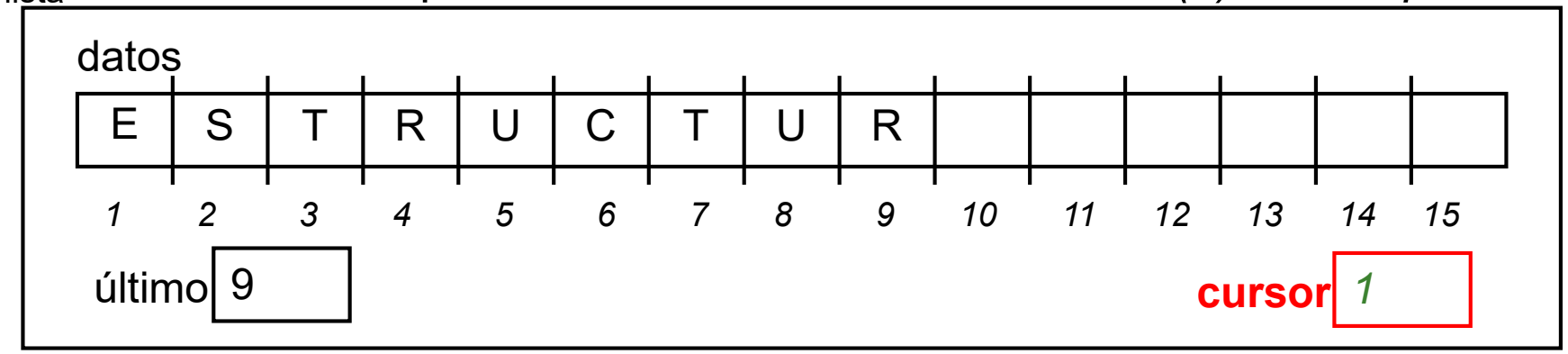
***1º) siguiente***

***2º) avanzar***

**fespec**

# Recorridos en lista con acceso por ambos extremos

- Implementación trivial con la representación estática vista en la lección:
  - cursor con un valor del índice (del elemento que toca "visitar")
    - **no deberá utilizarse en ninguna operación que no sea del iterador**
    - no compararlo en la operación de comparación (iguales)
  - todas las operaciones del iterador con *coste Θ(1) en tiempo*

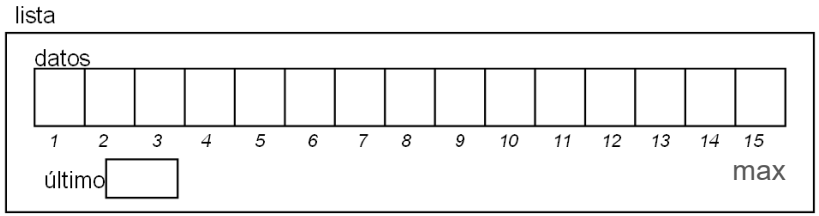


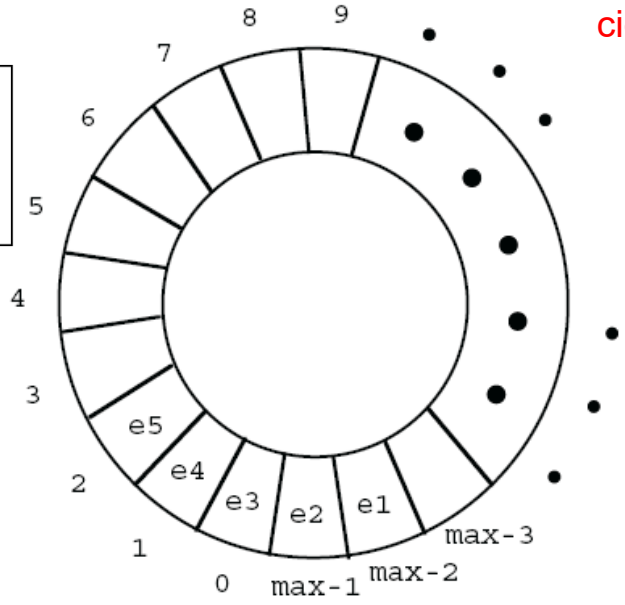

- Con una única operación (por ejemplo) llamémosle *siguienteYavanza*, se implementan 2 de la especificación: obtener *siguiente* elemento y (después) *avanza*
- Puede haber diferentes operaciones de recorrido de la secuencia (versiones de las operaciones de *iterador*):
  - en un sentido: *primero, existeSiguiente, siguiente, avanza, esÚltimo?, …*
  - o en otro: *último, existeAnterior, anterior, retrocede, esPrimero?, …*
  - o en ambos sentidos *(iterador bidireccional)*

# Implementación estática

- Alternativa de implementación:
  - ➢ las operaciones *añadirPrimero y borrarPrimero que tienen Θ(N) en tiempo:*
    - desplazarán elementos para evitar "huecos" (costes lineales)
    - Cómo evitarlo?

→ debe evitarse que el iterador cicle infinitamente …




```
anterior = max-3

último = 2
```

Considerar el "vector circular":

➢Operación de los índices del vector basándose en aritmética modular ( *mod max*)

➢Requiere campos recordando dónde está el ***último*** elemento de la lista y el ***anterior*** al primero

← *Ejercicio Propuesto*

¡CUIDADO! Evitar problema de "confusión" en la representación interna. . . .

→ no deben confundirse los casos de lista llena, lista vacía, lista con 1 elemento….

# Para pensarlo

- Ejercicios:
  - Implementación con un *"vector circular"*
  - Implementación en C++ del TAD lista con acceso por los extremos