TAD lista con acceso por los extremos. Implementación dinámica.

Lección 8

Listas genéricas con acceso por ambos extremos. Especificación. (lección 6)

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. (lección 6)

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 }

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

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 }

primero y último tienen que ser operaciones parciales

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

... {al final añadiremos las operaciones de un iterador}

Especificación de recorridos en listas genéricas (lección 6)

...

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}

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) }

fespec

Implementación dinámica en seudocódigo

```
módulo genérico listasGenéricasDinámicas
parámetro tipo elemento
exporta
   tipo lista
   procedimiento crear(sal L:lista)
   procedimiento añadirPrimero(ent e:elemento; e/s L:lista)
   procedimiento añadirÚltimo(e/s L:lista; ent e:elemento)
   función esVacía?(L:lista) devuelve booleano
   procedimiento borrarPrimero(e/s L:lista; sal error:bool)
   procedimiento borrarÚltimo(e/s L:lista; sal error: bool)
   procedimiento primero(ent L:lista;
                         sal e:elemento; sal error: bool)
   procedimiento último(ent L:lista;
                        sal e:elemento; sal error: bool)
   función longitud(L:lista) devuelve natural

   ...
```

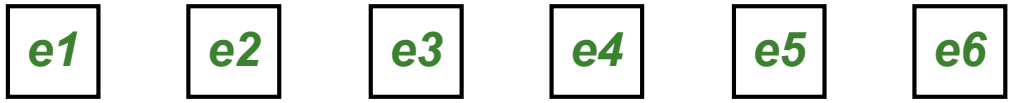
La capacidad de la lista no está limitada: *siempre* se podrá añadir (salvo que se agote la memoria disponible del ordenador)

Secuencias de elementos de un cierto tipo dispuestos en una dimensión

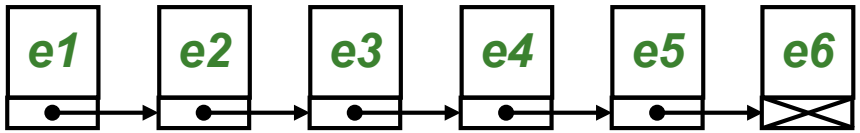
Ejemplo de secuencia:

e1 e2 e3 e4 e5 e6

¿Cómo almacenarla en memoria ocupando sólo el espacio necesario?

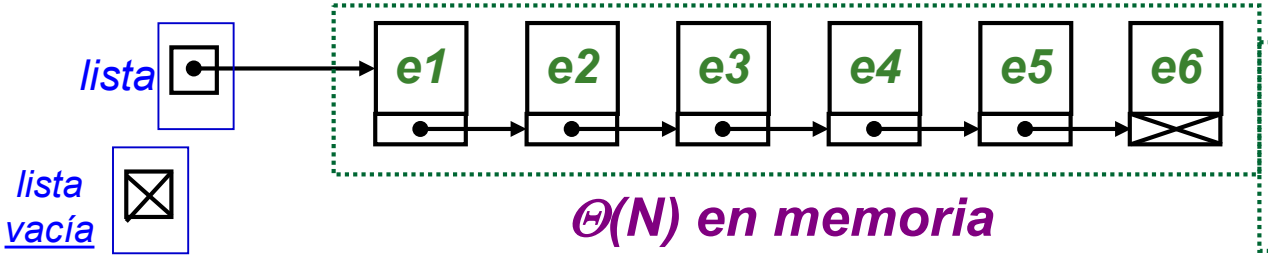


en datos creados dinámicamente en memoria, "reproduciendo" la secuencia…



Nodo o Celda

… con acceso al primero de la secuencia, y de cada elemento al siguiente



Θ(N) en memoria

A este tipo de estructura de datos también se le suele llamar "lista encadenada o enlazada"

¿Nos permitirá implementar las operaciones del *TAD lista con acceso por ambos extremos*? ¿con qué coste?

Implementación dinámica en seudocódigo

```
módulo genérico listasGenéricasDinámicas
parámetro tipo elemento
exporta
   tipo lista
   procedimiento crear(sal L:lista)
   procedimiento añadirPrimero(ent e:elemento; e/s L:lista)
   procedimiento añadirÚltimo(e/s L:lista; ent e:elemento)
   función esVacía?(L:lista) devuelve booleano
   procedimiento borrarPrimero(e/s L:lista; sal error:bool)
   procedimiento borrarÚltimo(e/s L:lista; sal error: bool)
   procedimiento primero(ent L:lista;
                         sal e:elemento; sal error: bool)
   procedimiento último(ent L:lista;
                        sal e:elemento; sal error: bool)
   función longitud(L:lista) devuelve natural

   ...
```

Implementación dinámica en seudocódigo

{ADEMÁS, se necesitará ofrecer operaciones (exportarlas):
- IGUALDAD, COPIAR (o duplicar),
- LIBERAR ...}

```
procedimiento copiar(sal lSal:lista; ent lEnt:lista)
{duplica la representación de lEnt en lSal (crea una copia completa o profunda)}
```

{ Si hay que añadir esta operación a la implementación del TAD, se podría decidir exigir que el tipo elemento cumpla una restricción: tener definida la operación de duplicar (copia profunda de datos de tipo elemento, en vez de copia superficial de elementos)}

```
función iguales?(l1,l2:lista) devuelve booleano
{devuelve verdad si las listas l1 y l2 tienen los mismos elementos, y en idénticas posiciones de la secuencia}
```

{ 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 (=) }

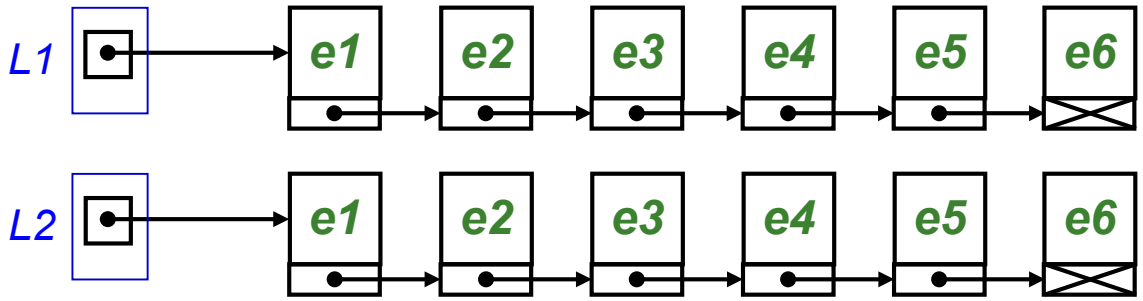
```
procedimiento liberar(e/s L:lista)
{libera toda la memoria ocupada por la lista L, dejando L vacía}
...
```

Implementación dinámica

Copia profunda (*deep copy*) versus copia superficial (*shallow copy*)

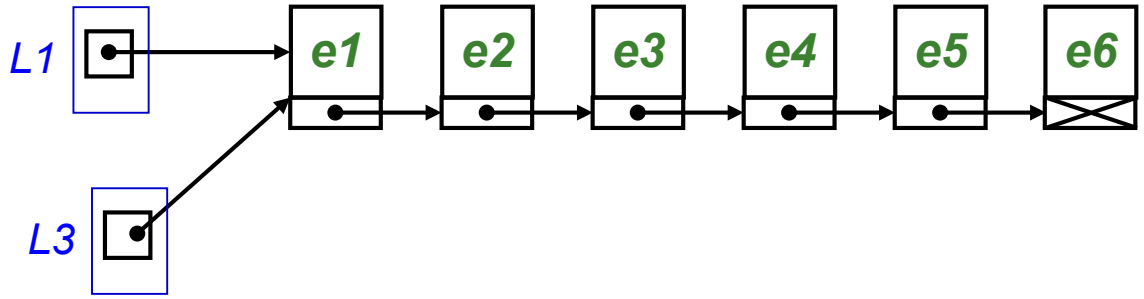
{Dibujado con *listas enlazadas simples*, pero es aplicable a cualquier estructura dinámica}

- Hacer copia profunda de L1 en L2:



Ambas listas podrán modificarse independientemente, i.e. sin afectar a la otra

- Hacer copia superficial de L1 en L3:

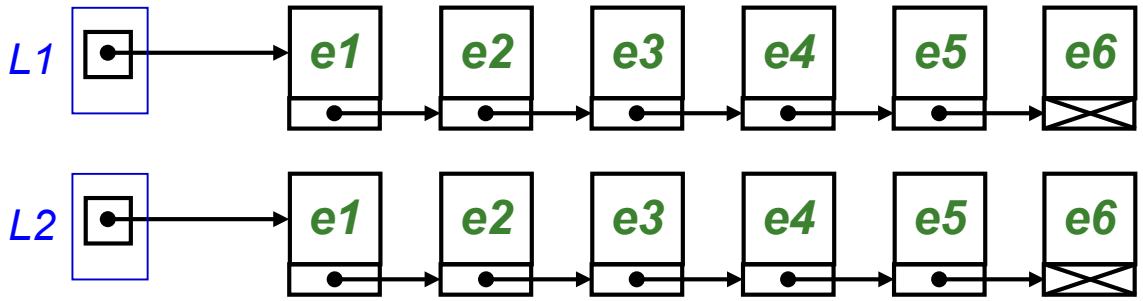


Las modificaciones hechas en una de las listas afectarán a la otra pudiendo corromperla (efectos colaterales muy graves)

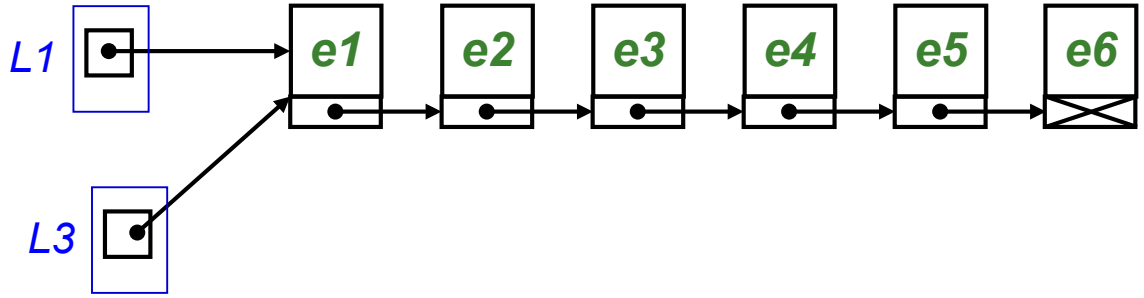
Hacer copias superficiales es muy eficiente (en tiempo y en uso de memoria) pero deben usarse con mucho cuidado

Implementación dinámica

```
función iguales?(l1,l2:lista) devuelve booleano
```
{devuelve verdad si las listas l1 y l2 tienen los mismos elementos, y en idénticas posiciones de la secuencia}



L1 y L2 son **iguales**: tienen los mismos elementos, y forman la misma secuencia



por el significado de la comparación de punteros:

L1 y L3 son iguales porque (sus punteros) apuntan al mismo sitio

{Dibujado con ***listas enlazadas simples***, pero este problema es trasladable a cualquier estructura dinámica}