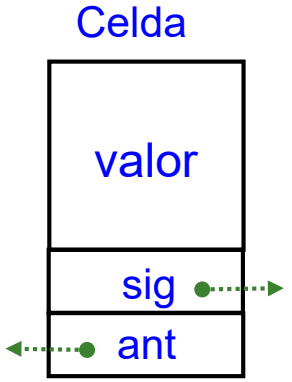
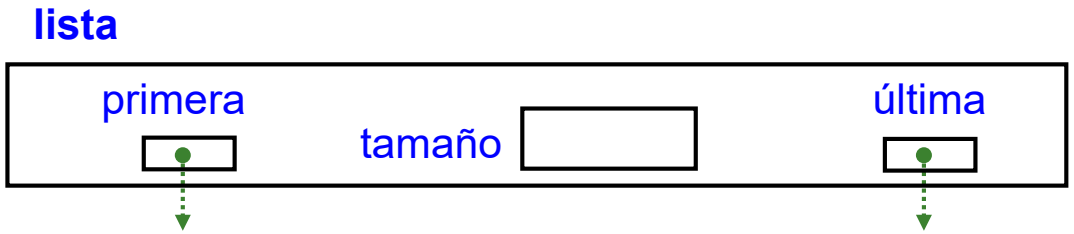
Implementación dinámica en seudocódigo

```
implementación    {dinámica, lista doblemente enlazada}
  tipos  punteroCelda = ↑Celda;

         Celda = registro
                    valor:elemento;
                    sig,ant:punteroCelda
                 freg


         lista = registro
                   primera,última:punteroCelda;
                   tamaño:natural
                 freg
```





...

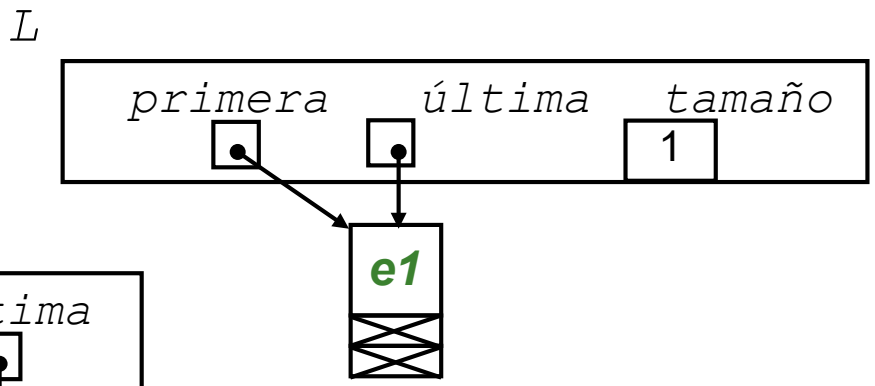
Implementación dinámica en seudocódigo

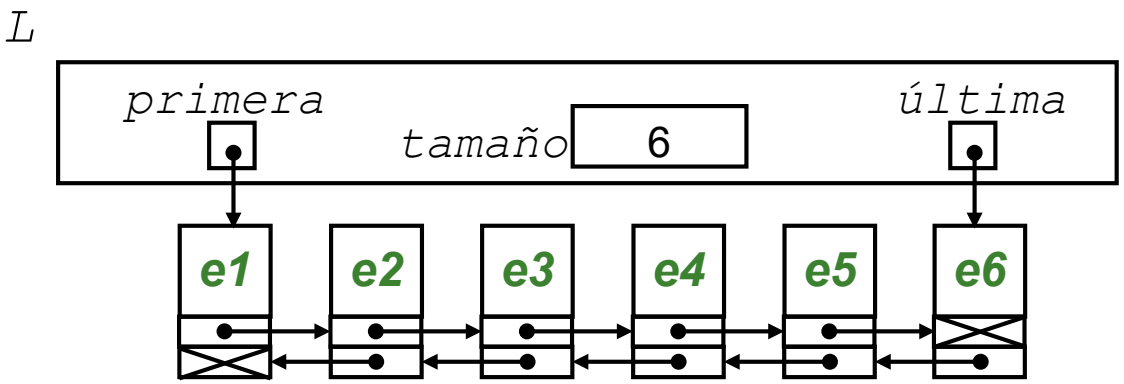
Casos a distinguir:

Θ(N) en memoria

- **lista vacía:**

- **lista NO vacía:**

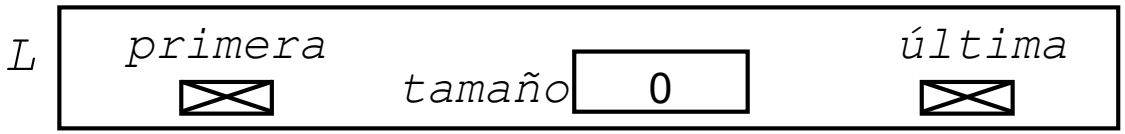




Implementación dinámica en seudocódigo

```
...
  procedimiento crear(sal L:lista)
  principio
    L.primera:=nil; L.última:=nil; L.tamaño:=0
  fin
 ...
```

Θ(1) en tiempo



{una lista vacía}

Implementación dinámica en seudocódigo

```
función esVacía?(L:lista) devuelve booleano
principio
  devuelve (L.primera=nil)
fin
```

Θ(1) en tiempo

```
función longitud(L:lista) devuelve natural
principio
  devuelve L.tamaño
fin
```

Θ(1) en tiempo

```
...
```

Implementación dinámica en seudocódigo

Θ(1) en tiempo

```
procedimiento primero(ent L:lista;
                      sal e:elemento; sal error: bool)
principio
  si L.primera=nil entonces  {caso de lista vacía}
    error:=verdad
  sino  {caso de lista NO vacía}
    error:=falso;
    e:=L.primera↑.valor
  fsi
fin
```

```
procedimiento último(ent L:lista;
                     sal e:elemento; sal error: bool)
{"simétrico" a primero:  intercambiar primera/última… }
...
```

Θ(1) en tiempo

Implementación dinámica en seudocódigo

```
procedimiento añadirPrimero(ent e:elemento; e/s L:lista)
variable pAux:punteroCelda
principio                                          Θ(1) en tiempo
  nuevoDato(pAux);
  pAux↑.valor:=e; paux↑.sig:=L.primera; pAux↑.ant:=nil;

  si L.primera=nil entonces {caso de lista vacía}
    L.última:=pAux

  sino  {caso de lista NO vacía}
    L.primera↑.ant:=pAux

  fsi;
  L.primera:=pAux;
  L.tamaño:=L.tamaño+1
fin
```

Implementación dinámica en seudocódigo

```
procedimiento añadirÚltimo(e/s L:lista; ent e:elemento)
{"simétrico" a añadirPrimero:  intercambiar sig/ant, primera/última… }
 variable pAux:punteroCelda
 principio
   nuevoDato(pAux);
   pAux↑.valor:=e; pAux↑.ant:=L.última; pAux↑.sig:=nil;

   si L.última=nil entonces {caso de lista vacía}
      L.primera:=pAux

   sino  {caso de lista NO vacía}
      L.última↑.sig:=pAux

   fsi;
   L.última:=pAux;
   L.tamaño:=L.tamaño+1
 fin
...
```

Θ(1) en tiempo

Implementación dinámica en seudocódigo

```
procedimiento borrarPrimero(e/s L:lista; sal error:bool)
   variable pAux:punteroCelda
 principio
    si L.primera=nil entonces  {caso de lista vacía}
      error:=verdad
    sino  {caso de lista NO vacía}
      error:=falso;
      pAux:=L.primera↑.sig;
      disponer(L.primera);
      L.primera:=pAux;
      si L.primera=nil entonces {caso de borrar el único elemento que había}
        L.última:=nil
      sino
        L.primera↑.ant:=nil;
      fsi;
      L.tamaño:=L.tamaño-1
    fsi
 fin
```

Θ(1) en tiempo

```
 procedimiento borrarÚltimo(e/s L:lista; sal error: bool)
 {"simétrico" a borrarPrimero:
     intercambiar sig/ant, primera/última… }
 ...
```

Θ(1) en tiempo

Implementación dinámica en seudocódigo

```
procedimiento copiar(sal lSal:lista; ent lEnt:lista)
{duplica la representación de lEnt en lSal (crea una
 copia completa o profunda)}
  variables pAux:punteroCelda; lSal2:lista
principio
  crear(lSal2);                                  {crear: Θ(1)tiempo}
  pAux:=lEnt.primera;
  mientrasQue pAux≠nil hacer
    añadirÚltimo(lSal2,pAux↑.valor);{añadirÚltimo: Θ(1)tiempo}
    pAux:=pAux↑.sig
  fmq;                                  {copia profunda, deep copy, de la lista}
  {copia superficial, shallow copy, de la lista }
  lSal:=lSal2
fin

...
```

Θ(N) en tiempo

```
función iguales?(L1,L2:lista) devuelve booleano
 {devuelve verdad si las listas L1 y L2 tienen los mismos
  elementos, y en idénticas posiciones}
 variables pAux1,pAux2:punteroCelda; iguales:booleano
  principio
    si esVacía?(L1) and esVacía?(L2) entonces
      devuelve verdad
    sino_si longitud(L1)≠longitud(L2) entonces
      devuelve falso
    sino
     {ninguna de las listas es vacía y tienen la misma longitud:}
      iguales:=verdad;
      pAux1:=L1.primera; pAux2:=L2.primera;
      mientrasQue iguales and pAux1≠nil hacer
        iguales:= (pAux1↑.valor=pAux2↑.valor);
        pAux1:=pAux1↑.sig;
        pAux2:=pAux2↑.sig
      fmq;
      devuelve iguales
    fsi
  fin
```

Θ(N) en caso peor, en tiempo

{ 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 (=) }

...

Implementación dinámica en seudocódigo

```
procedimiento liberar(e/s L:lista)
{libera toda la memoria ocupada por la lista L, dejando L vacía}
 variable pAux:punteroCelda
principio
    pAux:=L.primera;
    mientrasQue pAux≠nil hacer
      L.primera:=L.primera↑.sig;
      disponer(pAux);
      pAux:=L.primera
    fmq;
    crear(L)  {deja la lista L vacía: Θ(1) en tiempo}

 fin

{… añadir las operaciones del iterador…}

fin {del módulo}
```

Θ(N) en tiempo

Especificación de recorridos en listas genéricas (lección 6)

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

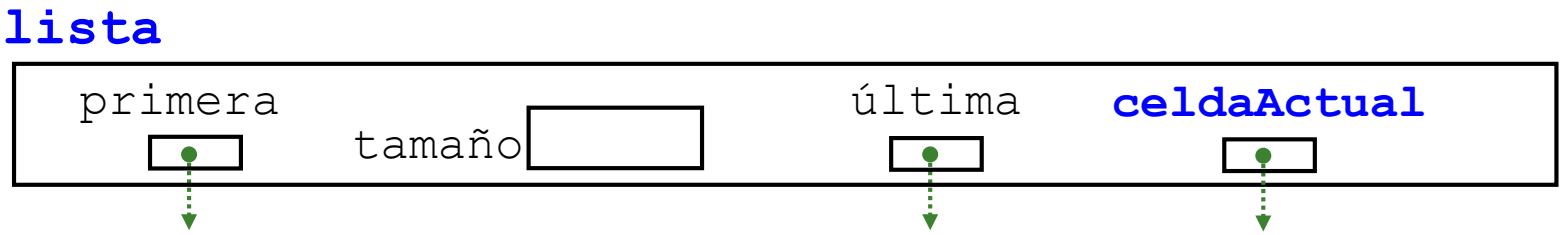
2º) avanzar

fespec

Iterador (interno)…

- Añadir al TAD las operaciones del iterador requiere que la representación interna de los valores del TAD sea:

```
lista = registro
          primera,última,celdaActual:punteroCelda;
          tamaño:natural
        freg
```



Iterador (interno). Implementación.

```
procedimiento iniciarIterador(e/s L:lista)
principio
  L.celdaActual:=L.primera                    Θ(1) en tiempo
fin

función existeSiguiente?(L:lista) devuelve booleano
principio
  devuelve (L.celdaActual≠nil)                Θ(1) en tiempo
fin

procedimiento siguienteYavanza(e/s L:lista;
                               sal e:elemento: sal error:booleano)
principio
  si existeSiguiente?(L) entonces             Θ(1) en tiempo
    error:=falso;
    e:=L.celdaActual↑.valor;{1°:elemento siguiente a devolver}
    L.celdaActual:=L.celdaActual↑.sig  {2°: avanzar}
  sino
    error:=verdad
  fsi
Fin
```

¡NINGUNA OTRA OPERACIÓN DEL TAD DEBE UTILIZAR NI LAS OPERACIONES DEL ITERADOR NI EL CAMPO DEL ITERADOR (celdaActual)!

- *Salvo POR SEGURIDAD en la **operación crear:** inicializar celdaActual a NIL*

En C++ (en un fichero *lista.hpp*)

```
// Interfaz del TAD. Pre-declaraciones:
template <typename Elemento> struct Lista;
// El tipo Elemento requerirá tener definida una función:
//    bool operator== (const Elemento& e1, const Elemento& e2);
//   {…devolverá true cuando…… devolverá false cuando…}
template <typename Elemento> void vacia(Lista<Elemento>& l);
template <typename Elemento> void anyadirPrimero(Lista<Elemento>& l, const Elemento& dato);
template <typename Elemento> void borrarPrimero(Lista<Elemento>& l);
template <typename Elemento> void anyadirUltimo(Lista<Elemento>& l, const Elemento& dato);
template <typename Elemento> void borrarUltimo(Lista<Elemento>& l);
template <typename Elemento> void primero(const Lista<Elemento>& l, Elemento& dato, bool& error);
template <typename Elemento> void ultimo(const Lista<Elemento>& l, Elemento& dato, bool& error);
template <typename Elemento> bool esVacia(const Lista<Elemento>& l);
template <typename Elemento> int longitud(const Lista<Elemento>& l);
template <typename Elemento> bool operator== (const Lista<Elemento>& l1,
                                              const Lista<Elemento>& l2);
template <typename Elemento> void duplicar(const  Lista<Elemento>& lOrigen,
                                           Lista<Elemento>& lDestino);
template <typename Elemento> void liberar(Lista<Elemento>& l);
template <typename Elemento> void iniciarIterador(Lista<Elemento>& l);
template <typename Elemento> bool haySiguiente(const Lista<Elemento>& l);
template <typename Elemento> bool siguienteYavanza(Lista<Elemento>& l, Elemento& dato);
  // sigue . . .
```

En la parte pública NO DEBEN aparecer detalles de implementación: No aparecen ni nodos, ni punteros a nodos,….

En C++ (en un fichero *lista.hpp*)

```
// Parte privada: Declaración de la representación interna
template <typename Elemento> struct Lista {

    friend void vacia<Elemento>(Lista<Elemento>& l);
    friend void anyadirPrimero<Elemento>(Lista<Elemento>& l, const Elemento& dato);
    friend void borrarPrimero<Elemento>(Lista<Elemento>& l);
    friend void anyadirUltimo<Elemento>(Lista<Elemento>& l, const Elemento& dato);
    friend void borrarUltimo<Elemento>(Lista<Elemento>& l);
    friend void primero<Elemento>(const Lista<Elemento>& l, Elemento& dato, bool& error);
    friend void ultimo<Elemento>(const Lista<Elemento>& l, Elemento& dato, bool& error);
    friend bool esVacia<Elemento>(const Lista<Elemento>& l);
    friend int longitud<Elemento>(const Lista<Elemento>& l);
    friend bool operator==<Elemento> (const Lista<Elemento>& l1, const Lista<Elemento>& l2);
    friend void duplicar<Elemento>(const Lista<Elemento>& lOrigen, Lista<Elemento>& lDestino);
    friend void liberar<Elemento>(Lista<Elemento>& l);
    friend void iniciarIterador<Elemento>(Lista<Elemento>& l);
    friend bool haySiguiente<Elemento>(const Lista<Elemento>& l);
    friend bool siguienteYavanza<Elemento>(Lista<Elemento>& l, Elemento& dato);
```

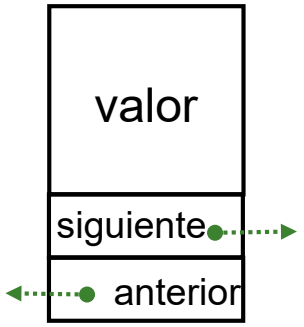
// sigue . . .

En C++ (en un fichero *lista.hpp*)

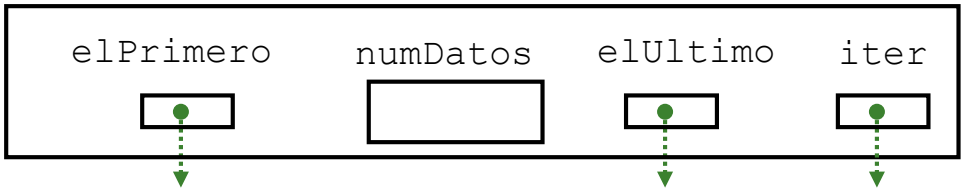
```
// Representación interna de los valores del TAD:
 private:
        struct Nodo {
            Elemento valor;
            Nodo* siguiente;
            Nodo* anterior;
        };

        Nodo* elPrimero ;
        Nodo* elUltimo;
        int numDatos;
        Nodo* iter;
};  //fin definición del struct Lista<Elemento>
    //Aquí fuera Nodo no está definido, pero si lo está Lista<Elemento>::Nodo  …
```

Lista<Elemento>::Nodo



Lista<Elemento>




```
// Parte privada: Implementación de las operaciones:
template<typename Elemento> void vacia(Lista<Elemento>& l) {
        l.numDatos = 0;
        l.elPrimero = nullptr;    l.elUltimo = nullptr;
        l.iter= nullptr;    // Inicialización del iterador, por seguridad…
}                                                          // sigue  . . .
```

En C++ (en un fichero *lista.hpp*)

```
template <typename Elemento> void anyadirPrimero(Lista<Elemento>& l, const Elemento& dato) {
  typename Lista<Elemento>::Nodo* aux= new typename Lista<Elemento>::Nodo;
   //asignamos valores a los campos del nuevo nodo:
   aux->valor=dato;
   aux->siguiente=l.elPrimero;
   aux->anterior=nullptr;
  //preparamos la lista y sus nodos:
   if (l.elUltimo==nullptr) {
      //la lista estaba vacía, el nuevo será primero y último elemento
      l.elUltimo=aux;
   }else{ //lista no vacía
      //el que era primero tendrá como anterior al nuevo nodo
      l.elPrimero->anterior=aux;
   }
   // ponemos el nuevo nodo como el primero de la lista:
   l.elPrimero=aux;
   l.numDatos++; // ahora la lista tiene un elemento más que antes
}
// etc etc implementación de las demás operaciones
...
```

```
//EQUIVALENTE a hacer:
// 1) declaración del puntero aux:
typename Lista<Elemento>::Nodo* aux;
// 2) reserva de memoria para un nodo, y asignación
// a aux para que apunte a la memoria reservada:
aux= new typename Lista<Elemento>::Nodo;
```