Análisis de Algoritmos y Estructuras de Datos

Tema 7: Tipo Abstracto de Datos Lista

Mª Teresa García Horcajadas José Fidel Argudo Argudo
Antonio García Domínguez Francisco Palomo Lozano

UCA
Universidad de Cádiz

Versión 4.0

Índice

Definición de Lista

Lista

Secuencia de elementos del mismo tipo, $(a_1, a_2, \ldots, a_n)$, cuya longitud, $n \geqslant 0$, es el número de elementos que contiene. Si $n = 0$, es decir, si la lista no tiene elementos, se denomina lista vacía.

Posición de un elemento

- Los elementos están ordenados linealmente según la posición que ocupa cada uno de ellos dentro de la lista.
- Todos los elementos, salvo el primero, tienen un único predecesor y todos, excepto el último, tienen un único sucesor.

Operaciones

Es posible acceder, insertar y suprimir elementos en cualquier posición de una lista.

Especificación del TAD *Lista*

Definición:

Una lista es una secuencia de elementos de un tipo determinado T

$$L = (a_1, a_2, \ldots, a_n)$$

cuya longitud es $n \geqslant 0$. Si $n = 0$, entonces es una lista vacía.

Posición Lugar que ocupa un elemento en la lista.
Los elementos están ordenados de forma lineal según las posiciones que ocupan. Todos los elementos, salvo el primero, tienen un único predecesor y todos, excepto el último, tienen un único sucesor.

Posición *fin()* Posición especial que sigue a la del último elemento y que nunca está ocupada por elemento alguno.

Especificación del TAD *Lista*

Operaciones:

Lista();

Postcondiciones: Crea una lista vacía.

bool vacia() const

Postcondiciones: Devuelve `true` si la lista está vacía, si no, `false`.

size_t tama() const

Postcondiciones: Devuelve la longitud de la lista.

void insertar(const T& x, posicion p)

Precondiciones: $L = (a_1, a_2, \ldots, a_n)$
$1 \leqslant p \leqslant n+1$

Postcondiciones: $L = (a_1, \ldots, a_{p-1}, x, a_p, \ldots, a_n)$

void eliminar(posicion p)

Precondiciones: $L = (a_1, a_2, \ldots, a_n)$
$1 \leqslant p \leqslant n$

Postcondiciones: $L = (a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_n)$

Especificación del TAD *Lista*

const T& elemento(posicion p) const

T& elemento(posicion p)

Precondiciones: $L = (a_1, a_2, \ldots, a_n)$
$1 \leqslant p \leqslant n$

Postcondiciones: Devuelve a_p, el elemento que ocupa la posición p.

posicion siguiente(posicion p) const

Precondiciones: $L = (a_1, a_2, \ldots, a_n)$
$1 \leqslant p \leqslant n$

Postcondiciones: Devuelve la posición que sigue a p.

posicion anterior(posicion p) const

Precondiciones: $L = (a_1, a_2, \ldots, a_n)$
$2 \leqslant p \leqslant n+1$

Postcondiciones: Devuelve la posición que precede a p.

Especificación del TAD *Lista*

posicion primera() const

Postcondiciones: Devuelve la primera posición de la lista. Si la lista está vacía, devuelve la posición *fin()*.

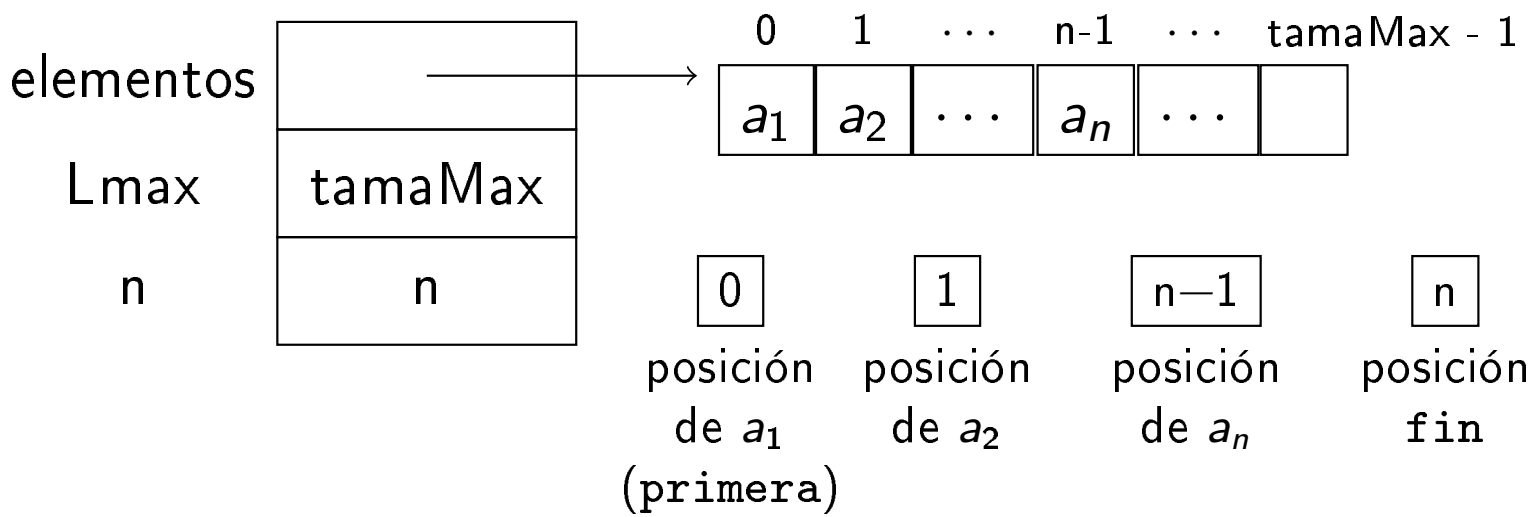
posicion fin() const

Postcondiciones: Devuelve la última posición de la lista, la siguiente a la del último elemento. Esta posición siempre está vacía, no existe ningún elemento que la ocupe.

Implementación vectorial pseudoestática

Implementación vectorial pseudoestática

Capacidad de la lista definida por el usuario del TAD mediante un parámetro del constructor.



Implementación vectorial pseudoestática (listavec.h)

```
1  // listavec.h
2  //
3  // clase Lista genérica cuya capacidad (parámetro de
4  //    entrada del constructor) puede ser distinta para
5  //    cada objeto de la clase.
6  //    Las variables externas de tipo posición, posteriores
7  //    a la posición p en la que se realiza una inserción
8  //    o eliminación, no cambian, pero sí los elementos
9  //    que se encuentran en dichas posiciones.

11 #ifndef LISTA_VEC_H
12 #define LISTA_VEC_H
13 #include <cstddef>    // size_t
14 #include <cassert>
```

Implementación vectorial pseudoestática (listavec.h)

```
16 template <typename T> class Lista {
17 public:
18    typedef size_t posicion;              // Posición de un elemento
19    explicit Lista(size_t tamaMax);       // Constructor, req. ctor. T()
20    bool vacia() const;
21    size_t tama() const;
22    size_t tamaMax() const;           // Requerida por la implementación
23    void insertar(const T& x, posicion p);
24    void eliminar(posicion p);
25    const T& elemento(posicion p) const;    // Lec. elto. en Lista const
26    T& elemento(posicion p);          // Lec/Esc elto. en Lista no-const
27    posicion siguiente(posicion p) const;
28    posicion anterior(posicion p) const;
29    posicion primera() const;
30    posicion fin() const;                 // Posición después del último
31    Lista(const Lista& L);                // Ctor. de copia, req. ctor. T()
32    Lista& operator =(const Lista& L); // Asig, req. ctor. T()
33    ~Lista();                             // Destructor
```

Implementación vectorial pseudoestática (listavec.h)

```
34 private:
35     T* elementos;       // Vector de elementos
36     size_t Lmax,        // Tamaño del vector
37            n;           // Longitud de la lista
38 };

40 // clase Lista genérica: vector pseudoestático.
41 //    Una lista de longitud n se almacena en celdas
42 //    consecutivas del vector, desde 0 hasta n-1.
43 //    La posición de un elemento es el índice de la celda
44 //    en que se almacena.
45 //
46 // Implementación de operaciones

48 template <typename T>
49 inline Lista<T>::Lista(size_t tamaMax) :
50     elementos(new T[tamaMax]),
51     Lmax(tamaMax),
52     n(0)
53 {}
```

Implementación vectorial pseudoestática (listavec.h)

```
55 template <typename T>
56 inline bool Lista<T>::vacia() const
57 {
58    return n == 0;
59 }

61 template <typename T>
62 inline size_t Lista<T>::tama() const
63 {
64    return n;
65 }

67 template <typename T>
68 inline size_t Lista<T>::tamaMax() const
69 {
70    return Lmax;
71 }
```

Implementación vectorial pseudoestática (listavec.h)

```
73 template <typename T>
74 void Lista<T>::insertar(const T& x, posicion p)
75 {
76    assert(p >= primera() && p <= fin());
77    assert(tama() < tamaMax());
78    // Desplazar los eltos. entre p y fin() a la siguiente posición
79    for (posicion q = fin(); q > p; --q)
80       elementos[q] = elementos[q-1];
81    elementos[p] = x;
82    ++n;
83 }
84 template <typename T>
85 void Lista<T>::eliminar(posicion p)
86 {
87    assert(p >= primera() && p < fin());
88    --n;
89    // Desplazar los eltos. entre p+1 y fin() a la posición anterior
90    for (posicion q = p; q < fin(); ++q)
91       elementos[q] = elementos[q+1];
92 }
```

Implementación vectorial pseudoestática (listavec.h)

```
94  template <typename T>
95  inline const T& Lista<T>::elemento(posicion p) const
96  {
97      assert(p >= primera() && p < fin());
98      return elementos[p];
99  }

101 template <typename T>
102 inline T& Lista<T>::elemento(posicion p)
103 {
104     assert(p >= primera() && p < fin());
105     return elementos[p];
106 }
```

Implementación vectorial pseudoestática (listavec.h)

```
108 template <typename T>
109 inline typename Lista<T>::posicion Lista<T>::siguiente(posicion p) const
110 {
111    assert(p >= primera() && p < fin());
112    return p+1;
113 }

115 template <typename T>
116 inline typename Lista<T>::posicion Lista<T>::anterior(posicion p) const
117 {
118    assert(p > primera() && p <= fin());
119    return p-1;
120 }

122 template <typename T>
123 inline typename Lista<T>::posicion Lista<T>::primera() const
124 { return 0; }

126 template <typename T>
127 inline typename Lista<T>::posicion Lista<T>::fin() const
128 { return n; }
```

Implementación vectorial pseudoestática (listavec.h)

```
130 // Constructor de copia
131 template <typename T>
132 Lista<T>::Lista(const Lista<T>& L) :
133    elementos(new T[L.Lmax]),
134    Lmax(L.Lmax),
135    n(L.n)
136 { // Copiar elementos
137    for (posicion p = primera(); p < fin(); ++p)
138       elementos[p] = L.elementos[p];
139 }
```

Implementación vectorial pseudoestática (listavec.h)

```
141 // Asignación de listas
142 template <typename T>
143 Lista<T>& Lista<T>::operator =(const Lista<T>& L)
144 {
145    if (this != &L) {    // Evitar autoasignación
146       // Destruir el vector y crear uno nuevo si es necesario
147       if (Lmax != L.Lmax) {
148          T* p = elementos;
149          elementos = new T[L.Lmax]; // Si new falla, *this no cambia.
150          Lmax = L.Lmax;
151          delete[] p;
152       }
153       n = L.n;
154       for (posicion p = primera(); p < fin(); ++p)
155          elementos[p] = L.elementos[p];
156    }
157    return *this;
158 }
```

Implementación vectorial pseudoestática (listavec.h)

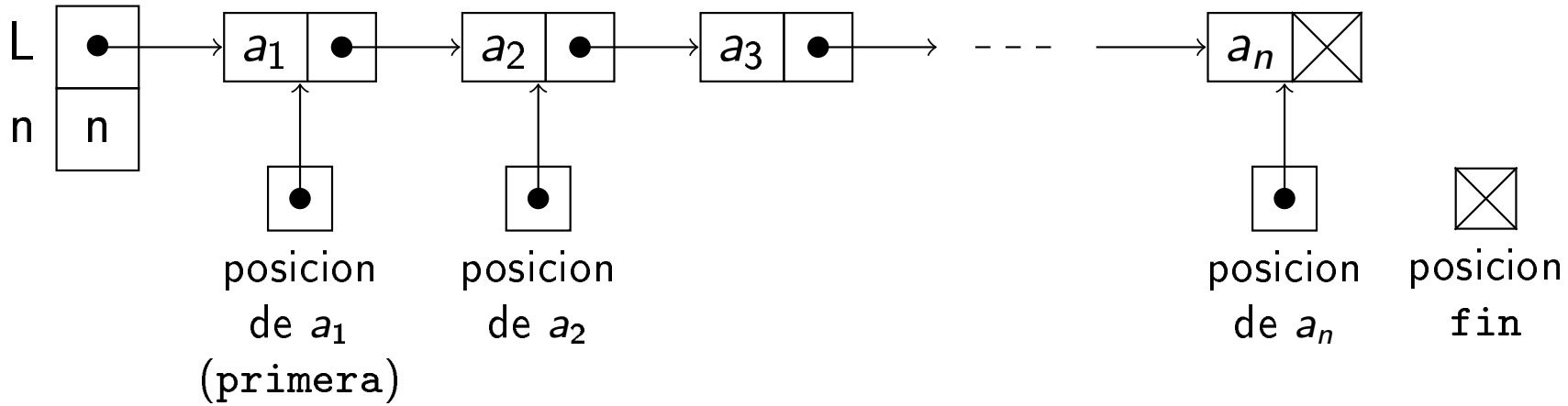
```
160 // Destructor
161 template <typename T>
162 inline Lista<T>::~Lista()
163 {
164    delete[] elementos;
165 }

167 #endif // LISTA_VEC_H
```

Implementación mediante una estructura enlazada

Estructura dinámica

El tamaño de la estructura de datos varía durante la ejecución con el tamaño de la lista, pero los enlaces ocupan espacio adicional.



Representación de posiciones

Posición de un elemento Puntero al nodo que lo contiene

Primera posición Puntero al primer nodo de la estructura

Última posición (*fin()*) Puntero nulo, el almacenado en el último nodo.

Implementación mediante una estructura enlazada (listaenla0.h)

```
1  template <typename T> class Lista {
2     struct nodo;    // Declaración adelantada privada
3  public:
4     typedef nodo* posicion;    // Posición de un elemento
5     // ...
6  private:
7     struct nodo {
8        T elto;
9        nodo* sig;
10       nodo(const T& e, nodo* p = nullptr) : elto(e), sig(p) {}
11    };

13    nodo* L;    // Lista enlazada de nodos
14    size_t n;   // Longitud de la lista
15 };

17 template <typename T>
18 inline Lista<T>::Lista() : L(nullptr), n(0) {}
```

Implementación mediante una estructura enlazada (listaenla0.h)

```
20 template <typename T>
21 inline typename Lista<T>::posicion Lista<T>::siguiente(posicion p) const
22 {
23    assert(p != fin());
24    return p->sig;
25 }

27 template <typename T>
28 typename Lista<T>::posicion Lista<T>::anterior(posicion p) const
29 {
30    assert(p != primera());
31    posicion q = primera();
32    while (q->sig != p) q = q->sig;
33    return q;
34 }

36 template <typename T>
37 inline typename Lista<T>::posicion Lista<T>::primera() const
38 { return L; }

40 template <typename T>
41 inline typename Lista<T>::posicion Lista<T>::fin() const
42 { return nullptr; }
```

Implementación mediante una estructura enlazada (listaenla0.h)

```
44 template <typename T>
45 void Lista<T>::insertar(const T& x, posicion& p)
46 {
47    if (p == primera())
48       p = L = new nodo(x, p);
49    else { // Inserción en cualquier otra posición, incluso fin()
50       posicion q = anterior(p);
51       p = q->sig = new nodo(x, p);
52    }
53    ++n;
54    // El nuevo nodo con x queda en la posición p
55 }
```

Implementación mediante una estructura enlazada (listaenla0.h)

```
57 template <typename T>
58 void Lista<T>::eliminar(posicion& p)
59 {
60    assert(p != fin());
61    if (p == primera()) {
62       L = p->sig;
63       delete p;
64       p = primera();
65    }
66    else {
67       posicion q = anterior(p);
68       q->sig = p->sig;
69       delete p;
70       p = q->sig;
71    }
72    --n;
73    // El nodo siguiente queda en la posición p
74 }
```

Implementación mediante una estructura enlazada

Inserción y eliminación de elementos

1. Los algoritmos de inserción y eliminación son de orden $\Theta(n)$ en el promedio y en el peor caso. Hay que recorrer la lista desde el inicio hasta la posición anterior a p.
2. En la inserción es posible evitar el recorrido copiando en el nuevo nodo el que se encuentra en la posición p y colocando ahí el elemento a insertar junto al enlace al nuevo nodo.
3. El recorrido al eliminar se puede evitar copiando en el nodo de la posición p el que le sigue y suprimiendo este.

Inserción en una lista enlazada. Versión 2. (lisaenla0.h)

```
1  template <typename T>
2  void Lista<T>::insertar(const T& x, posicion& p)
3  {
4     if (p != fin())
5        *p = nodo(x, new nodo(*p));
6     else // Inserción al final
7        if (vacia())
8           p = L = new nodo(x);
9        else {
10          posicion q = anterior(fin());
11          p = q->sig = new nodo(x);
12       }
13    ++n;
14 }
```

Eliminación en una lista enlazada. Versión 2. (lisaenla0.h)

```
16 template <typename T> void Lista<T>::eliminar(posicion& p)
17 {
18    assert(p != fin());
19    if (p->sig != fin()) { // *p no es el último
20       posicion q = p->sig;
21       *p = *q;
22       delete q;
23    }
24    else // Eliminar el último
25       if (p == primera()) {
26          delete p;
27          p = L = fin(); // Lista vacía
28       }
29       else {
30          posicion q = anterior(p);
31          delete p;
32          p = q->sig = fin();
33       }
34    --n;
35 }
```

Implementación mediante una estructura enlazada

Inserción y eliminación en una lista enlazada. Versión 2

1. Ahora la inserción y la eliminación son $\Theta(1)$ en el promedio, pero siguen teniendo un peor caso $\Theta(n)$. Hay que recorrer toda la lista para añadir un nuevo nodo al final, así como para eliminar el último.
2. Debemos considerar la copia adicional de un elemento. Si el elemento es grande, el tiempo de recorrido ahorrado puede no compensar el tiempo extra de copia.
3. En definitiva, ambos algoritmos son de orden $\Theta(n)$ en el caso peor y en los demás casos puede que la ganancia de tiempo no sea significativa con la segunda versión.

Implementación mediante una estructura enlazada

Inserción y eliminación de elementos

1. El parámetro de tipo `posicion` de estas operaciones se pasa por referencia, porque un nuevo elemento ocupará dicha posición al finalizar.
2. No se cumple totalmente con la especificación del TAD, porque este parámetro se debe pasar por valor.
3. Por ello, el uso del TAD Lista con esta implementación provocará errores de compilación que no se producirán con otras implementaciones.

Implementación mediante una estructura enlazada

Errores de compilación

```
1 Lista<int> l;
2 l.insertar(5, l.fin());
3 l.insertar(3, l.primera());
4 l.insertar(4,
5   l.anterior(l.fin()));
6 l.eliminar(l.primera());
```

Correcto

```
1 Lista<int> l;
2 Lista<int>::posicion p;
3 p = l.fin();
4 l.insertar(5, p);
5 l.insertar(3, p);
6 p = l.anterior(l.fin());
7 l.insertar(4, p);
8 p = l.primera();
9 l.eliminar(p);
```

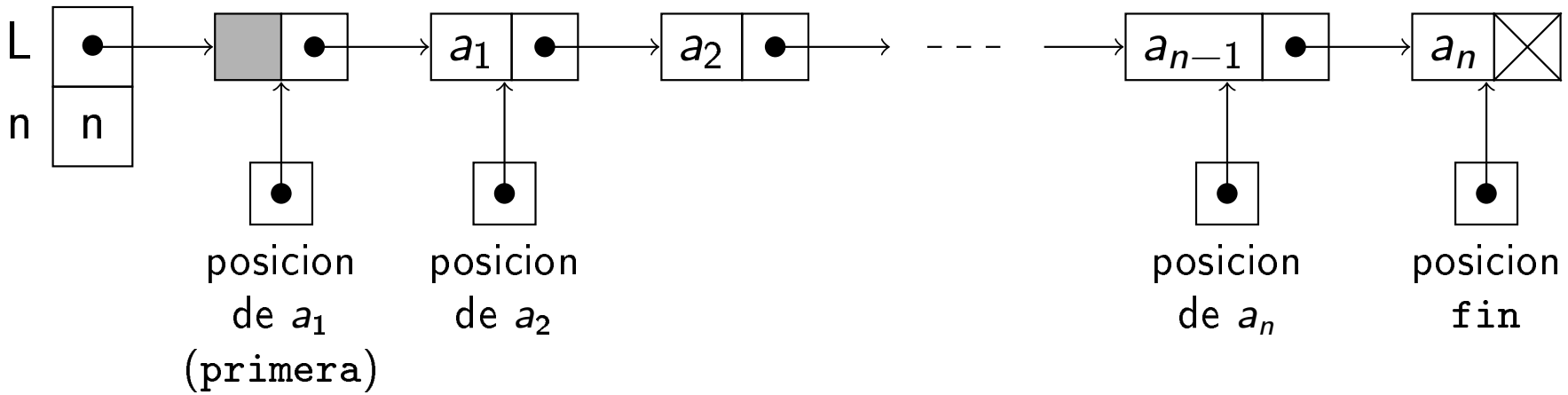
Incumplimiento de la especificación

El código de la izquierda es correcto según la especificación del TAD, sin embargo produce errores de compilación porque no se pueden pasar por referencia a *insertar()* y *eliminar()* las posiciones devueltas por *fin()*, *primera()*, *anterior()* o *siguiente()*.

Implementación con una estructura enlazada con cabecera

Modificación de la representación del TAD Lista

- Para solventar el incumplimiento de la especificación y la ineficiencia de las inserciones y eliminaciones cambiamos el modo de representar las posiciones: la posición de un elemento es representada por un puntero al nodo anterior.
- El cambio de representación del tipo `posicion` nos lleva a introducir un nodo cabecera para facilitar las operaciones en la primera posición.



Impl. con una estr. enlazada con cabecera (listaenla1.h)

```
1  template <typename T> class Lista {
2     struct nodo;    // Declaración adelantada privada
3  public:
4     typedef nodo* posicion;    // Posición de un elemento
5     Lista();                   // Constructor, req. ctor. T()
6     // ...
7  private:
8     struct nodo {
9        T elto;
10       nodo* sig;
11       nodo(const T& e = T(), nodo* p = nullptr) : elto(e), sig(p) {}
12    };

14    nodo* L;    // Lista enlazada con cabecera
15    size_t n;   // Longitud de la lista
16 };

18 template <typename T>
19 inline Lista<T>::Lista() :
20    L(new nodo), // Crear cabecera
21    n(0)
22 {}
```

Impl. con una estr. enlazada con cabecera (listaenla1.h)

```
24 template <typename T>
25 inline void Lista<T>::insertar(const T& x, posicion p)
26 {
27     p->sig = new nodo(x, p->sig);
28     ++n;
29     // El nuevo nodo con x queda en la posición p
30 }

32 template <typename T>
33 inline void Lista<T>::eliminar(posicion p)
34 {
35     assert(p->sig != nullptr); // p no es fin()
36     nodo* q = p->sig;
37     p->sig = q->sig;
38     delete q;
39     --n;
40     // El nodo siguiente queda en la posición p
41 }
```

Impl. con una estr. enlazada con cabecera (listaenla1.h)

```
43 template <typename T> inline
44 typename Lista<T>::posicion Lista<T>::siguiente(posicion p) const
45 {
46    assert(p->sig != nullptr); // p != fin()
47    return p->sig;
48 }

50 template <typename T>
51 typename Lista<T>::posicion Lista<T>::anterior(posicion p) const
52 {
53    assert(p != primera());
54    posicion q = primera();
55    while (q->sig != p) q = q->sig;
56    return q;
57 }
```

Impl. con una estr. enlazada con cabecera (listaenla1.h)

```
59 template <typename T>
60 inline typename Lista<T>::posicion Lista<T>::primera() const
61 { return L; }

63 template <typename T>
64 typename Lista<T>::posicion Lista<T>::fin() const
65 {
66    posicion p = primera();
67    while (p->sig != nullptr) p = p->sig;
68    return p;
69 }
```

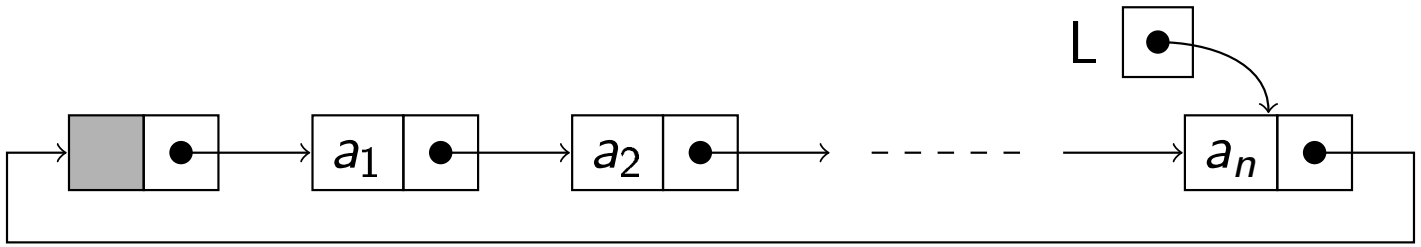
Implementación con una estructura enlazada con cabecera

Eficiencia

1. Las operaciones *anterior()* y *fin()* son $\Theta(n)$.
2. El resto de operaciones del TAD son $\Theta(1)$.

Estructura enlazada circular

La operación *fin()* se puede hacer de $\Theta(1)$ sin alterar la eficiencia de las demás operaciones y sin usar espacio adicional, representando una lista mediante una estructura enlazada circular que proporcione acceso directo al último nodo.



Impl. con una estr. enlazada circular con cab. (listaenla.h)

```
1  // clase Lista genérica de capacidad ilimitada.
2  //    Después de una inserción o eliminación en una
3  //    posición p, las variables externas de tipo posición
4  //    posteriores a p continúan representado las
5  //    posiciones de los mismos elementos.

7  #ifndef LISTA_ENLA_H
8  #define LISTA_ENLA_H
9  #include <cstddef>    // size_t
10 #include <cassert>

12 template <typename T> class Lista {
13    struct nodo;    // Declaración adelantada privada
14 public:
15    typedef nodo* posicion;                    // Posición de un elemento
16    Lista();                                   // Constructor, req. ctor. T()
17    bool vacia() const;
18    size_t tama() const;
19    void insertar(const T& x, posicion p);
20    void eliminar(posicion p);
21    const T& elemento(posicion p) const;       // Lec. elto. en Lista const
22    T& elemento(posicion p);                   // Lec/Esc elto. en Lista no-const
```

Impl. con una estr. enlazada circular con cab. (listaenla.h)

```
23      posicion siguiente(posicion p) const;
24      posicion anterior(posicion p) const;
25      posicion primera() const;
26      posicion fin() const;                          // Posición después del último
27      Lista(const Lista& Lis);                       // Ctor. de copia, req. ctor. T()
28      Lista& operator =(const Lista& Lis);           // Asig. de listas
29      ~Lista();                                      // Destructor
30  private:
31      struct nodo {
32          T elto;
33          nodo* sig;
34          nodo(const T& e = T(), nodo* p = nullptr): elto(e), sig(p) {}
35      };

37      nodo* L;    // Lista enlazada circular con cabecera, ptro. al último
38      size_t n;   // Longitud de la lista

40      void copiar(const Lista& Lis);
41  };
```

Impl. con una estr. enlazada circular con cab. (listaenla.h)

```
43 template <typename T>
44 inline Lista<T>::Lista() :
45    L(new nodo), // Crear cabecera
46    n(0)
47 {
48    L->sig = L; // Estructura circular
49 }

51 template <typename T>
52 inline bool Lista<T>::vacia() const
53 {
54    return n == 0; // Alternativa: return primera() == fin();
55 }

57 template <typename T>
58 inline size_t Lista<T>::tama() const
59 {
60    return n;
61 }
```

Impl. con una estr. enlazada circular con cab. (listaenla.h)

```
63 template <typename T>
64 inline void Lista<T>::insertar(const T& x, posicion p)
65 {
66    p->sig = new nodo(x, p->sig);
67    if (p == fin())
68       L = p->sig;    // Nuevo último
69    ++n;
70 }

72 template <typename T>
73 inline void Lista<T>::eliminar(posicion p)
74 {
75    assert(p != fin());
76    posicion q = p->sig;
77    if (q == fin())
78       L = p; // El nuevo último es el penúltimo
79    p->sig = q->sig;
80    delete q;
81    --n;
82 }
```

Impl. con una estr. enlazada circular con cab. (listaenla.h)

```
84 template <typename T>
85 inline const T& Lista<T>::elemento(posicion p) const
86 {
87    assert(p != fin());
88    return p->sig->elto;
89 }

91 template <typename T>
92 inline T& Lista<T>::elemento(posicion p)
93 {
94    assert(p != fin());
95    return p->sig->elto;
96 }
```

Impl. con una estr. enlazada circular con cab. (listaenla.h)

```
 98 template <typename T>
 99 inline typename Lista<T>::posicion Lista<T>::siguiente(posicion p) const
100 {
101    assert(p != fin());
102    return p->sig;
103 }

105 template <typename T>
106 typename Lista<T>::posicion Lista<T>::anterior(posicion p) const
107 {
108    assert(p != primera());
109    posicion q = primera();
110    while (q->sig != p) q = q->sig;
111    return q;
112 }

114 template <typename T>
115 inline typename Lista<T>::posicion Lista<T>::primera() const
116 { return L->sig; }

118 template <typename T>
119 inline typename Lista<T>::posicion Lista<T>::fin() const
120 { return L; }
```

Impl. con una estr. enlazada circular con cab. (listaenla.h)

```
122 // Constructor de copia
123 template <typename T>
124 inline Lista<T>::Lista(const Lista& Lis) : Lista()
125 {
126    copiar(Lis);
127 }

129 // Asignación de listas
130 template <typename T>
131 Lista<T>& Lista<T>::operator =(const Lista& Lis)
132 {
133    if (this != &Lis) {  // Evitar autoasignación
134       while (!vacia()) eliminar(primera());
135       copiar(Lis);
136    }
137    return *this;
138 }
```

Impl. con una estr. enlazada circular con cab. (listaenla.h)

```
140 // Destructor: vacía la lista y destruye el nodo cabecera
141 template <typename T> Lista<T>::~Lista()
142 {
143    nodo* p;
144    while (L != L->sig) {
145       p = L->sig;
146       L->sig = p->sig;
147       delete p;
148    }
149    delete L;
150 }
```

Impl. con una estr. enlazada circular con cab. (listaenla.h)

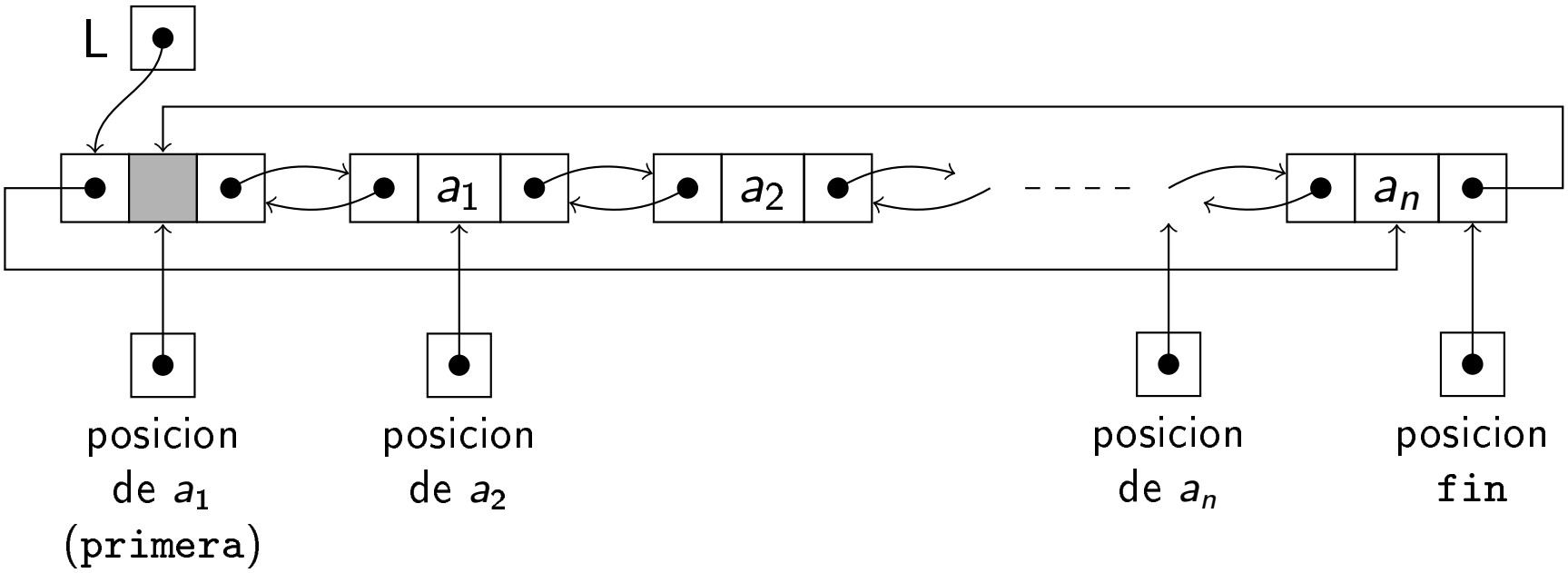
```
152 // Método privado
153 template <typename T>
154 void Lista<T>::copiar(const Lista& Lis)
155 // Pre: *this está vacía.
156 // Post: *this es copia de Lis.
157 {
158    for (nodo* p = Lis.L->sig; p != Lis.L; p = p->sig, ++n)
159       L = L->sig = new nodo(p->sig->elto, L->sig);
160 }

162 #endif // LISTA_ENLA_H
```

Impl. con estr. circular doblemente enlazada con cabecera

Estructura doblemente enlazada

A costa de doblar la cantidad de punteros, se optimiza la eficiencia de la operación *anterior()*.



Impl. con estr. cir. doblemente enlazada con cab. (listadoble.h)

```
1  #ifndef LISTA_DOBLE_H
2  #define LISTA_DOBLE_H
3  #include <cstddef>    // size_t
4  #include <cassert>

6  template <typename T> class Lista {
7     struct nodo;    // Declaración adelantada privada
8  public:
9     typedef nodo* posicion;                    // Posición de un elemento
10    Lista();                                   // Constructor, req. ctor. T()
11    bool vacia() const;
12    size_t tama() const;
13    void insertar(const T& x, posicion p);
14    void eliminar(posicion p);
15    const T& elemento(posicion p) const;       // Lec. elto. en Lista const
16    T& elemento(posicion p);                   // Lec/Esc elto. en Lista no-const
17    posicion siguiente(posicion p) const;
18    posicion anterior(posicion p) const;
19    posicion primera() const;
20    posicion fin() const;                      // Posición después del último
21    Lista(const Lista& Lis);                   // Ctor. de copia, req. ctor. T()
22    Lista& operator =(const Lista& Lis);       // Asig. de listas
23    ~Lista();                                  // Destructor
```

Impl. con estr. cir. doblemente enlazada con cab. (listadoble.h)

```
24 private:
25     struct nodo {
26         T elto;
27         nodo *ant, *sig;
28         nodo(const T& e = T(), nodo* a = nullptr, nodo* s = nullptr)
29             : elto(e), ant(a), sig(s) {}
30     };

32     nodo* L;    // Lista doblemente enlazada circular con cabecera
33     size_t n;   // Longitud de la lista

35     void copiar(const Lista& Lis);
36 };
```

Impl. con estr. cir. doblemente enlazada con cab. (listadoble.h)

```
38 template <typename T>
39 inline Lista<T>::Lista() :
40    L(new nodo), // Crear cabecera
41    n(0)
42 {
43    L->ant = L->sig = L; // Estructura circular
44 }

46 template <typename T>
47 inline bool Lista<T>::vacia() const
48 {
49    return n == 0; // Alternativa: return primera() == fin();
50 }

52 template <typename T>
53 inline size_t Lista<T>::tama() const
54 {
55    return n;
56 }
```

Impl. con estr. cir. doblemente enlazada con cab. (listadoble.h)

```
58 template <typename T>
59 inline void Lista<T>::insertar(const T& x, posicion p)
60 {
61    p->sig = p->sig->ant = new nodo(x, p, p->sig);
62    ++n;
63 }

65 template <typename T>
66 inline void Lista<T>::eliminar(posicion p)
67 {
68    assert(p != fin());
69    nodo* q = p->sig;
70    p->sig = q->sig;
71    p->sig->ant = p;
72    delete q;
73    --n;
74 }
```

Impl. con estr. cir. doblemente enlazada con cab. (listadoble.h)

```
76 template <typename T>
77 inline const T& Lista<T>::elemento(posicion p) const
78 {
79    assert(p != fin());
80    return p->sig->elto;
81 }

83 template <typename T>
84 inline T& Lista<T>::elemento(posicion p)
85 {
86    assert(p != fin());
87    return p->sig->elto;
88 }
```

Impl. con estr. cir. doblemente enlazada con cab. (listadoble.h)

```
 90 template <typename T>
 91 inline typename Lista<T>::posicion Lista<T>::siguiente(posicion p) const
 92 {
 93    assert(p != fin());
 94    return p->sig;
 95 }

 97 template <typename T>
 98 inline typename Lista<T>::posicion Lista<T>::anterior(posicion p) const
 99 {
100    assert(p != primera());
101    return p->ant;
102 }

104 template <typename T>
105 inline typename Lista<T>::posicion Lista<T>::primera() const
106 { return L; }

108 template <typename T>
109 inline typename Lista<T>::posicion Lista<T>::fin() const
110 { return L->ant; }
```

Impl. con estr. cir. doblemente enlazada con cab. (listadoble.h)

```
112 // Constructor de copia
113 template <typename T>
114 inline Lista<T>::Lista(const Lista& Lis) : Lista()
115 {
116    copiar(Lis);
117 }

119 // Asignación de listas
120 template <typename T>
121 Lista<T>& Lista<T>::operator =(const Lista& Lis)
122 {
123    if (this != &Lis) {  // Evitar autoasignación
124       while (!vacia()) eliminar(primera());
125       copiar(Lis);
126    }
127    return *this;
128 }
```

Impl. con estr. cir. doblemente enlazada con cab. (listadoble.h)

```
130 // Destructor: vacía la lista y destruye el nodo cabecera
131 template <typename T> Lista<T>::~Lista()
132 {
133    nodo* p;
134    while (L != L->sig) {
135       p = L->sig;
136       L->sig = p->sig;
137       delete p;
138    }
139    delete L;
140 }
```

Impl. con estr. cir. doblemente enlazada con cab. (listadoble.h)

```
142 // Método privado
143 template <typename T>
144 void Lista<T>::copiar(const Lista& Lis)
145 // Pre: *this está vacía.
146 // Post: *this es copia de Lis.
147 {
148    for (nodo* p = Lis.L->sig; p != Lis.L; p = p->sig, ++n)
149       L->ant = L->ant->sig = new nodo(p->elto, L->ant, L);
150 }

152 #endif // LISTA_DOBLE_H
```

Comparación de representaciones del TAD *Lista*

Eficiencia de operaciones

Operación	**Vectorial** (listavec.h)	**Enlazada** (listaenla.h)	**Doblemente enlazada** (listadoble.h)
elemento	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
insertar eliminar	$\Theta(n)$		
primera fin siguiente	$\Theta(1)$		
anterior		$\Theta(n)$	

Comparación de representaciones del TAD *Lista*

Representación vectorial

- Una lista tiene capacidad limitada, establecida al construirla.
- Estimar la capacidad adecuada puede ser un problema. Fijarla en un valor excesivo supone desaprovechamiento del espacio no utilizado. Tener que aumentarla a posteriori implica un coste en tiempo para reubicar el contenido en una lista de mayor capacidad.
- El espacio ocupado es el justo para almacenar el máximo de elementos, sin coste adicional.
- La inserción y eliminación requieren desplazar elementos de la lista. De ahí que la eficiencia de estas operaciones sea lineal respecto al número de elementos desplazados (distancia hasta el final de la lista).
- El acceso secuencial bidireccional es de eficiencia óptima, ya que obtener el predecesor y sucesor son operaciones elementales.

Comparación de representaciones del TAD *Lista*

Representación enlazada

- Estructura diseñada para inserciones y eliminaciones rápidas en cualquier posición, eficiencia $\Theta(1)$.
- La capacidad no está limitada.
- Cada elemento está enlazado con su sucesor, lo que supone un coste de almacenamiento adicional y que el acceso secuencial hacia adelante sea muy eficiente.
- El acceso al predecesor es poco eficiente, orden lineal.

Representación doblemente enlazada

- En comparación con la representación enlazada se duplica el espacio adicional para almacenar cada elemento, pues se enlaza con predecesor y sucesor; pero se consigue acceso secuencial bidireccional de máxima eficiencia.

Comparación de representaciones del TAD *Lista*

Selección de representación

La estructura de datos para el TAD *Lista* se elige según los requisitos de la aplicación siguiendo criterios de eficiencia:

- La vectorial es la más eficiente en espacio si la longitud de la lista se mantiene cercana a su capacidad, por lo que es la mejor opción si además dicha longitud es estable (pocas inserciones y eliminaciones). También proporciona el acceso secuencial bidireccional más rápido.
- Cuando sea difícil estimar el tamaño máximo y para listas de longitud muy variable, es decir, con muchas inserciones y eliminaciones, es preferible una estructura enlazada, o doblemente enlazada si el acceso al predecesor es una operación crítica.