

Análisis de Algoritmos y Estructuras de Datos

Tema 7: Tipo Abstracto de Datos Lista

M^a Teresa García Horcajadas José Fidel Argudo Argudo
Antonio García Domínguez Francisco Palomo Lozano



Versión 1.0



Índice

- 1 Definición del TAD Lista
- 2 Especificación del TAD Lista
- 3 Implementación del TAD Lista

Definición de Lista

Lista

Secuencia de elementos del mismo tipo, (a_1, a_2, \dots, a_n) , cuya **longitud**, $n \geq 0$, 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

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

cuya **longitud** es $n \geq 0$. 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 y devuelve una lista vacía.

void insertar(const T& x, posicion p)

Precondiciones: $L = (a_1, a_2, \dots, a_n)$

$$1 \leq p \leq n + 1$$

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

void eliminar(posicion p)

Precondiciones: $L = (a_1, a_2, \dots, a_n)$

$$1 \leq p \leq n$$

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

Especificación del TAD *Lista*

const T& elemento(posicion p) const

T& elemento(posicion p)

Precondiciones: $L = (a_1, a_2, \dots, a_n)$
 $1 \leq p \leq n$

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

posicion buscar(const T& x) const

Postcondiciones: Devuelve la posición de la primera ocurrencia de x en la lista. Si x no se encuentra, devuelve la posición *fin()*.

Especificación del TAD *Lista*

posicion siguiente(posicion p) const

Precondiciones: $L = (a_1, a_2, \dots, a_n)$

$$1 \leq p \leq n$$

Postcondiciones: Devuelve la posición que sigue a p .

posicion anterior(posicion p) const

Precondiciones: $L = (a_1, a_2, \dots, a_n)$

$$2 \leq p \leq n + 1$$

Postcondiciones: Devuelve la posición que precede a p .

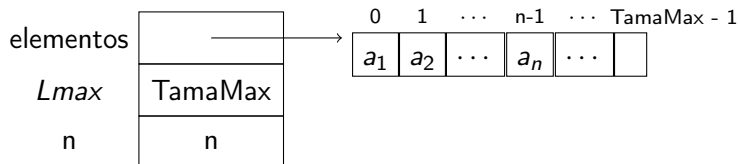
posicion primera() const

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

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

12 #ifndef LISTA_VEC_H
13 #define LISTA_VEC_H
14 #include <cassert>
```

Implementación vectorial pseudoestática

```
16 template <typename T>
17 class Lista {
18 public:
19     typedef int posicion; // posición de un elto
20     explicit Lista(size_t TamaMax); // constructor
21     Lista(const Lista<T>& l); // ctor. de copia
22     Lista<T>& operator =(const Lista<T>& l); // asignación entre
        listas
23     void insertar(const T& x, posicion p);
24     void eliminar(posicion p);
25     const T& elemento(posicion p) const; // acceso a elto, lectura
26     T& elemento(posicion p); // acceso a elto, lectura/escritura
27     posicion buscar(const T& x) const; // requiere operador ==
        para el tipo T
28     posicion siguiente(posicion p) const;
29     posicion anterior(posicion p) const;
30     posicion primera() const;
31     posicion fin() const; // posición después del último
32     ~Lista(); // destructor
```

Implementación vectorial pseudoestática

```
33 private:
34     T *elementos; // vector de elementos
35     int Lmax; // tamaño del vector
36     int n; // longitud de la lista
37 };

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

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

Implementación vectorial pseudoestática

```
54 template <typename T>
55 Lista<T>::Lista(const Lista<T>& l) :
56     elementos(new T[l.Lmax]),
57     Lmax(l.Lmax),
58     n(l.n)
59 {
60     for (Lista<T>::posicion p = 0; p < n; p++) // copiar el vector
61         elementos[p] = l.elementos[p];
62 }
```

Implementación vectorial pseudoestática

```
64 template <typename T>
65 Lista<T>& Lista<T>::operator =(const Lista<T>& l)
66 {
67     if (this != &l) { // evitar autoasignación
68         // Destruir el vector y crear uno nuevo si es necesario
69         if (Lmax != l.Lmax) {
70             delete[] elementos;
71             Lmax = l.Lmax;
72             elementos = new T[Lmax];
73         }
74         // Copiar el vector
75         n = l.n;
76         for (Lista<T>::posicion p = 0; p < n; p++)
77             elementos[p] = l.elementos[p];
78     }
79     return *this;
80 }
```

Implementación vectorial pseudoestática

```
82 template <typename T>
83 void Lista<T>::insertar(const T& x, Lista<T>::posicion p)
84 {
85     assert(p >= 0 && p <= n); // posición válida
86     assert(n < Lmax); // lista no llena
87     for (Lista<T>::posicion q = n; q > p; q--)
88         // desplazar los eltos. en p, p+1, ...
89         elementos[q] = elementos[q-1]; // a la siguiente posición
90     elementos[p] = x;
91     n++;
92 }
93 template <typename T>
94 void Lista<T>::eliminar(Lista<T>::posicion p)
95 {
96     assert(p >= 0 && p < n); // posición válida
97     n--;
98     for (Lista<T>::posicion q = p; q < n; q++)
99         //desplazar los eltos. en p+1, p+2, ...
100         elementos[q] = elementos[q+1]; // a la posición anterior
101 }
```

Implementación vectorial pseudoestática

```
103 template <typename T> inline
104 const T& Lista<T>::elemento(Lista<T>::posicion p) const
105 {
106     assert(p >= 0 && p < n); // posición válida
107     return elementos[p];
108 }

110 template <typename T>
111 inline T& Lista<T>::elemento(Lista<T>::posicion p)
112 {
113     assert(p >= 0 && p < n); // posición válida
114     return elementos[p];
115 }
```

Implementación vectorial pseudoestática

```
117 template <typename T>
118 typename Lista<T>::posicion Lista<T>::buscar(const T& x)
    const
119 {
120     Lista<T>::posicion q = 0;
121     bool encontrado = false;
122     while (q < n && !encontrado)
123         if (elementos[q] == x)
124             encontrado = true;
125         else q++;
126     return q;
127 }

129 template <typename T>
130 inline typename Lista<T>::posicion Lista<T>::siguiente(Lista<
    T>::posicion p) const
131 {
132     assert(p >= 0 && p < n); // posición válida
133     return p+1;
134 }
```


Implementación vectorial pseudoestática

```
136 template <typename T>
137 inline typename Lista<T>::posicion Lista<T>::anterior(Lista<T
    >::posicion p) const
138 {
139     assert(p > 0 && p <= n); // posición válida
140     return p-1;
141 }
142 template <typename T>
143 inline typename Lista<T>::posicion Lista<T>::primera() const
144 { return 0; }

146 template <typename T>
147 inline typename Lista<T>::posicion Lista<T>::fin() const
148 { return n; }

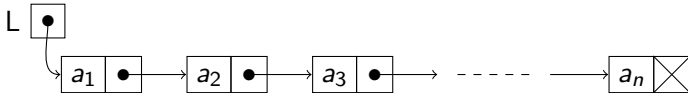
150 template <typename T>
151 inline Lista<T>::~Lista()
152 { delete[] elementos; }

154 #endif // LISTA_VEC_H
```

Implementación mediante una estructura enlazada

Estructura dinámica

El tamaño de la estructura de datos varía en tiempo de ejecución con el tamaño de la lista. A cambio se ocupa espacio adicional con los enlaces.



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 almacenado en el último nodo de la estructura, o sea, un puntero nulo.

Implementación mediante una estructura enlazada

```
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
6      void insertar(const T& x, posicion& p);
7      void eliminar(posicion& p);
8      // .....
9  private:
10     struct nodo {
11         T elto;
12         nodo* sig;
13         nodo(T e, nodo* p = 0): elto(e), sig(p) {}
14     };

16     nodo* L; // lista enlazada de nodos
17 };
```

Implementación mediante una estructura enlazada

```
19 template <typename T>
20 inline Lista<T>::Lista() : L(0) {}

21 template <typename T>
22 void Lista<T>::insertar(const T& x, Lista<T>::posicion& p)
23 {
24     nodo* q;

25
26     if (p == L) // inserción al principio
27         p = L = new nodo(x, p);
28     else { // inserción en cualquier otra posición, incluso fin
29         // recorrer la lista hasta el nodo q anterior a p
30         for (q = L; q->sig != p; q = q->sig);
31         p = q->sig = new nodo(x, p);
32     }
33     // el nuevo nodo con x queda en la posición p
34 }
```

Implementación mediante una estructura enlazada

```
36 template <typename T>
37 void Lista<T>::eliminar(Lista<T>::posicion& p)
38 {
39     nodo* q;

41     assert(p); // p no es fin
42     if (p == L) { /* primera posición */
43         L = p->sig;
44         delete p;
45         p = L;
46     }
47     else {
48         // recorrer la lista hasta el nodo q anterior a p
49         for (q = L; q->sig != p; q = q->sig);
50         q->sig = p->sig;
51         delete p;
52         p = q->sig;
53     }
54     // el nodo siguiente queda en la posición p
55 }
```

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 el nodo 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 en este el elemento que se inserta.
- 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

```
1  template <typename T>
2  void Lista<T>::insertar(const T& x, Lista<T>::posicion& p)
3  {
4      nodo* q;

6      if (p) { // p no es fin
7          q = new nodo(p->elto, p->sig); // copia *p en *q
8          p->elto = x;
9          p->sig = q;
10     }
11     else // inserción al final
12         if (L == 0) // lista vacía
13             p = L = new nodo(x);
14         else {
15             // recorrer la lista hasta el último nodo
16             for (q = L; q->sig; q = q->sig);
17             p = q->sig = new nodo(x);
18         }
19 }
```

Eliminación en una lista enlazada. Versión 2

```
1  template <typename T> void Lista<T>::eliminar(posicion& p)
2  {
3      nodo* q;
4      assert(p); // p no es fin
5      if (p->sig) { // *p no es el último
6          q = p->sig;
7          *p = *q;
8          delete q;
9      }
10     else // eliminar el último
11         if (p == L) { // primera posición
12             delete p;
13             p = L = 0; // fin
14         }
15         else {
16             // recorrer la lista hasta el penúltimo nodo
17             for (q = L; q->sig != p; q = q->sig);
18             delete p;
19             p = q->sig = 0; // fin
20         }
21 }
```


Implementación mediante una estructura enlazada

Inserción y eliminación en una lista enlazada. Versión 2

- ➊ 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.
- ➋ 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.
- ➌ 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 posición 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 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.primer());  
4 l.insertar(4,  
5     l.anterior(l.fin()));  
6 l.eliminar(l.primer());
```

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.primer();  
9 l.eliminar(p);
```

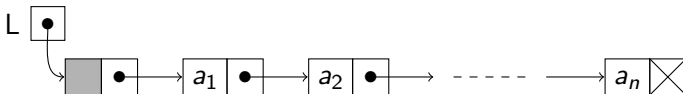
Incumplimiento de la especificación

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()*, *primer()*, *anterior()*, *siguiente()* o *buscar()*.

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.



Representación de posiciones

Posición de un elemento Puntero al nodo anterior

Primera posición Puntero al nodo cabecera

Última posición (*fin()*) Puntero al último nodo de la estructura.

Implementación con una estructura enlazada con cabecera

```
1  #ifndef LISTA_ENLA_H
2  #define LISTA_ENLA_H
3  #include <cassert>

5  template <typename T> class Lista {
6      struct nodo; // declaración adelantada privada
7  public:
8      typedef nodo* posicion; // posición de un elemento
9      Lista(); // constructor, requiere ctor. T()
10     Lista(const Lista<T>& l); // ctor. de copia, requiere ctor. T()
11     Lista<T>& operator =(const Lista<T>& l); // asignación de
        listas
12     void insertar(const T& x, posicion p);
13     void eliminar(posicion p);
14     const T& elemento(posicion p) const; // acceso a elto, lectura
15     T& elemento(posicion p); // acceso a elto, lectura/escritura
```

Implementación con una estructura enlazada con cabecera

```
16  posicion buscar(const T& x) const; // T requiere operador ==
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(); // destructor
22 private:
23     struct nodo {
24         T elto;
25         nodo* sig;
26         nodo(const T& e, nodo* p = 0): elto(e), sig(p) {}
27     };

29     nodo* L; // lista enlazada de nodos

31     void copiar(const Lista<T>& l);
32 };
```

Implementación con una estructura enlazada con cabecera

```
34 // Método privado
35 template <typename T>
36 void Lista<T>::copiar(const Lista<T> &l)
37 {
38     L = new nodo(T()); // crear el nodo cabecera
39     nodo* q = L;
40     for (nodo* r = l.L->sig; r; r = r->sig) {
41         q->sig = new nodo(r->elto);
42         q = q->sig;
43     }
44 }
```

Implementación con una estructura enlazada con cabecera

```
46 template <typename T>
47 inline Lista<T>::Lista() : L(new nodo(T())) // crear cabecera
48 {}

50 template <typename T>
51 inline Lista<T>::Lista(const Lista<T>& l)
52 {
53     copiar(l);
54 }

56 template <typename T>
57 Lista<T>& Lista<T>::operator =(const Lista<T>& l)
58 {
59     if (this != &l) { // evitar autoasignación
60         this->~Lista(); // vaciar la lista actual
61         copiar(l);
62     }
63     return *this;
64 }
```


Implementación con una estructura enlazada con cabecera

```
66 template <typename T> inline
67 void Lista<T>::insertar(const T& x, Lista<T>::posicion p)
68 {
69     p->sig = new nodo(x, p->sig);
70     // el nuevo nodo con x queda en la posición p
71 }

73 template <typename T>
74 inline void Lista<T>::eliminar(Lista<T>::posicion p)
75 {
76     assert(p->sig); // p no es fin
77     nodo* q = p->sig;
78     p->sig = q->sig;
79     delete q;
80     // el nodo siguiente queda en la posición p
81 }
```

Implementación con una estructura enlazada con cabecera

```
83 template <typename T> inline
84 const T& Lista<T>::elemento(Lista<T>::posicion p) const
85 {
86     assert(p->sig); // p no es fin
87     return p->sig->elto;
88 }

90 template <typename T>
91 inline T& Lista<T>::elemento(Lista<T>::posicion p)
92 {
93     assert(p->sig); // p no es fin
94     return p->sig->elto;
95 }
```

Implementación con una estructura enlazada con cabecera

```
97  template <typename T>
98  typename Lista<T>::posicion
99      Lista<T>::buscar(const T& x) const
100 {
101     nodo* q = L;
102     bool encontrado = false;
103     while (q->sig && !encontrado)
104         if (q->sig->elto == x)
105             encontrado = true;
106         else q = q->sig;
107     return q;
108 }

110 template <typename T> inline
111 typename Lista<T>::posicion
112     Lista<T>::siguiente(Lista<T>::posicion p) const
113 {
114     assert(p->sig); // p no es fin
115     return p->sig;
116 }
```

Implementación con una estructura enlazada con cabecera

```
118 template <typename T>
119 typename Lista<T>::posicion
120     Lista<T>::anterior(Lista<T>::posicion p) const
121 {
122     nodo* q;

124     assert(p != L); // p no es la primera posición
125     for (q = L; q->sig != p; q = q->sig);
126     return q;
127 }

129 template <typename T>
130 inline typename Lista<T>::posicion Lista<T>::primera() const
131 {
132     return L;
133 }
```

Implementación con una estructura enlazada con cabecera

```
135 template <typename T>
136 typename Lista<T>::posicion Lista<T>::fin() const
137 {
138     nodo* p;
139     for (p = L; p->sig; p = p->sig);
140     return p;
141 }

143 // Destructor: destruye el nodo cabecera y vacía la lista
144 template <typename T> Lista<T>::~~Lista()
145 {
146     nodo* q;
147     while (L) {
148         q = L->sig;
149         delete L;
150         L = q;
151     }
152 }

154 #endif // LISTA_ENLA_H
```

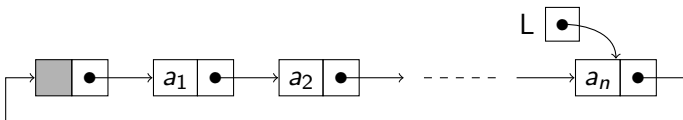
Implementación con una estructura enlazada con cabecera

Eficiencia

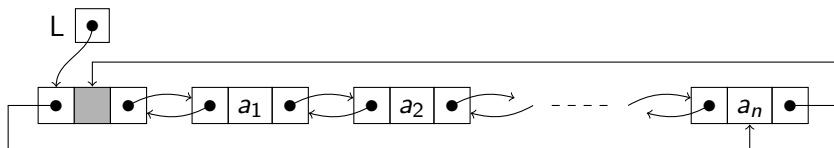
- 1 Las operaciones *buscar()*, *anterior()* y *fin()* son $\Theta(n)$.
- 2 El resto de operaciones del TAD son $\Theta(1)$.

Estructura enlazada circular

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



Impl. con una estructura doblemente enlazada con cabecera



Representación de posiciones

Posición de un elemento Puntero al nodo anterior

Primera posición Puntero al nodo cabecera

Última posición (*fin()*) Puntero al último nodo de la estructura.

Impl. con una estructura doblemente enlazada con cabecera

```
1  #ifndef LISTA_DOBLE_H
2  #define LISTA_DOBLE_H
3  #include <cassert>

5  template <typename T> class Lista {
6      struct nodo; // declaración adelantada privada
7  public:
8      typedef nodo* posicion; // posición de un elemento
9      Lista(); // constructor, requiere ctor. T()
10     Lista(const Lista<T>& l); // ctor. de copia, requiere ctor. T()
11     Lista<T>& operator =(const Lista<T>& l); // asignación entre
        listas
12     void insertar(const T& x, posicion p);
13     void eliminar(posicion p);
14     const T& elemento(posicion p) const; // acceso a elto, lectura
15     T& elemento(posicion p); // acceso a elto, lectura/escritura
```


Impl. con una estructura doblemente enlazada con cabecera

```
16  posicion buscar(const T& x) const; // T requiere operador ==
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(); // destructor
22 private:
23     struct nodo {
24         T elto;
25         nodo *ant, *sig;
26         nodo(const T& e, nodo* a = 0, nodo* s = 0) :
27             elto(e), ant(a), sig(s) {}
28     };

30     nodo* L; // lista doblemente enlazada de nodos

32     void copiar(const Lista<T>& l);
33 };
```

Impl. con una estructura doblemente enlazada con cabecera

```
35 // Método privado
36 template <typename T>
37 void Lista<T>::copiar(const Lista<T> &l)
38 {
39     L = new nodo(T()); // crear el nodo cabecera
40     L->ant = L->sig = L; // estructura circular
41     // Copiar elementos de l
42     for (nodo* q = l.L->sig; q != l.L; q = q->sig)
43         L->ant = L->ant->sig = new nodo(q->elto, L->ant, L);
44 }
```

Impl. con una estructura doblemente enlazada con cabecera

```
46 template <typename T>
47 inline Lista<T>::Lista() : L(new nodo(T())) // crear cabecera
48 {
49     L->ant = L->sig = L; // estructura circular
50 }

52 template <typename T>
53 inline Lista<T>::Lista(const Lista<T>& l)
54 { copiar(l); }

56 template <typename T>
57 Lista<T>& Lista<T>::operator =(const Lista<T>& l)
58 {
59     if (this != &l) { // evitar autoasignación
60         this->~Lista(); // vaciar la lista actual
61         copiar(l);
62     }
63     return *this;
64 }
```

Impl. con una estructura doblemente enlazada con cabecera

```
66 template <typename T> inline
67 void Lista<T>::insertar(const T& x, Lista<T>::posicion p)
68 {
69     p->sig = p->sig->ant = new nodo(x, p, p->sig);
70     // el nuevo nodo con x queda en la posición p
71 }

73 template <typename T>
74 inline void Lista<T>::eliminar(Lista<T>::posicion p)
75 {
76     assert(p->sig != L); // p no es fin
77     nodo* q = p->sig;
78     p->sig = q->sig;
79     p->sig->ant = p;
80     delete q;
81     // el nodo siguiente queda en la posición p
82 }
```

Impl. con una estructura doblemente enlazada con cabecera

```
84 template <typename T> inline
85 const T& Lista<T>::elemento(Lista<T>::posicion p) const
86 {
87     assert(p->sig != L); // p no es fin
88     return p->sig->elto;
89 }

91 template <typename T>
92 inline T& Lista<T>::elemento(Lista<T>::posicion p)
93 {
94     assert(p->sig != L); // p no es fin
95     return p->sig->elto;
96 }
```

Impl. con una estructura doblemente enlazada con cabecera

```
98 template <typename T>
99 typename Lista<T>::posicion
100 Lista<T>::buscar(const T& x) const
101 {
102     nodo* q = L;
103     bool encontrado = false;
104     while (q->sig != L && !encontrado)
105         if (q->sig->elto == x)
106             encontrado = true;
107         else q = q->sig;
108     return q;
109 }
```

Impl. con una estructura doblemente enlazada con cabecera

```
111 template <typename T> inline
112 typename Lista<T>::posicion
113     Lista<T>::siguiente(Lista<T>::posicion p) const
114 {
115     assert(p->sig != L); // p no es fin
116     return p->sig;
117 }

119 template <typename T> inline
120 typename Lista<T>::posicion
121     Lista<T>::anterior(Lista<T>::posicion p) const
122 {
123     assert(p != L); // p no es la primera posición
124     return p->ant;
125 }
```

Impl. con una estructura doblemente enlazada con cabecera

```
127 template <typename T>
128 inline typename Lista<T>::posicion Lista<T>::primera() const
129 {
130     return L;
131 }

133 template <typename T>
134 inline typename Lista<T>::posicion Lista<T>::fin() const
135 {
136     return L->ant;
137 }
```


Impl. con una estructura doblemente enlazada con cabecera

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

152 #endif // LISTA_DOBLE_H
```