



## 4. Contenedores. T.D.A Lineales

### 4.1 Contenedores lineales

Los tipos de datos que se caracterizan por ser contenedores, almacenan colecciones de datos de un mismo tipo. Hasta ahora hemos profundizado en algún contenedor como es el vector dinámico. Además este se caracteriza por ser lineal ya que los datos se disponen de manera secuencial uno detrás de otro. Los contenedores lineales son tipos de datos muy interesantes ya que se comportan de manera idéntica independiente del tipo de dato que almacenan. Por lo tanto son tipos de datos candidatos a ser implementados como clases plantillas. Veremos en este tema que dependiendo de las posibilidades de acceso, inserción y borrado adoptaremos un contenedor lineal u otro. Así los aspectos a resaltar son:

**Contenedores** : estructuras que almacenan datos de un mismo tipo base. Ej: vectores y listas

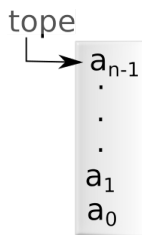
**Lineales** : contienen una secuencia de elementos dispuestos en una dimensión  $\{a_0, a_1, \dots, a_n\}$

En este capítulo los contenedores lineales sobre los que vamos a tratar son:

1. Pilas
2. Colas
3. Colas con prioridad
4. Listas

### 4.2 Pila

1. **Especificación.** Contiene una secuencia de elementos  $\{a_0, a_1, \dots, a_{n-1}\}$  en la que las pilas están optimizadas para realizar inserciones, consultas y borrados por sólo uno de los extremos (Estructura LIFO: *Primero en entrar es el primero en salir*(**L**ast **I**nput **F**irst **O**utput))



- Las consultas se realizan sobre  $a_{n-1}$
- Los borrados se hacen sobre  $a_{n-1}$
- Las inserciones se hacen sobre  $a_{n-1}$
- No se puede acceder a la pila por otro lado que no sea el tope

## 2. Operación:

- *Tope*: consulta el elemento del tope
- *Vacía*: devuelve true si la pila no tiene ningún elemento
- *Quitar(pop)*: elimina el elemento en el tope
- *Poner(put)*: inserta un nuevo elemento en el tope

## 3. Implementación:

- Basada en vectores
- Basada en celdas enlazadas

### 4.2.1 Vectores estáticos

Una primera aproximación a la implementación de una Pila es usando un vector estático, con un tamaño predefinido. Este tamaño debe ser suficiente para albergar el máximo número de datos. Si en promedio el número de datos alojados es muy pequeño frente al total de posibles elementos a albergar, existirá un exceso de memoria innecesario.

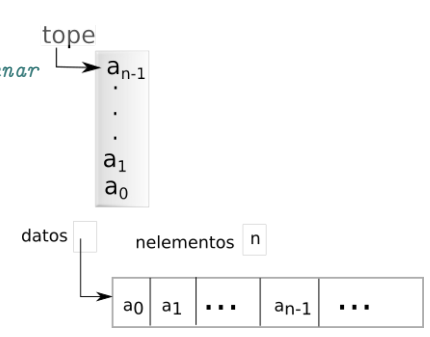
Por otro lado, todas las operaciones se ejecutan en tiempo constante. En este caso, el *tope* ocupa la posición  $nelementos - 1$ . Así, cada vez que se quiera consultar, con la función *Tope*, se devuelve el elemento situado en esa posición. Para realizar una nueva inserción, se añade el nuevo elemento en la posición  $nelementos$  del vector y se incrementa  $nelementos$  en 1. Cuando se quiera borrar, usando la función *Quitar*, basta con decrementar  $nelementos$  en 1.

Por lo tanto, esta representación es útil cuando se tiene una buena estimación del máximo de elementos que puedes llegar a almacenar y en promedio no se desaprovecha mucha memoria.

```

1  #include <cassert>
2
3  template <class T>
4  class Pila {
5      private:
6          T datos[200]; //Cantidad real que nunca vamos a llenar
7          int nelementos;
8
9      public:
10         Pila () {
11             nelementos = 0;
12         }
13
14         T Tope () const {
15             //el tope seria nelementos-1
16             assert (nelementos > 0); //nos aseguramos de que hay, al menos, un elemento
17             return datos[nelementos-1];
18         }
19
20         void Quitar () {
21             assert (nelementos > 0);
22             nelementos--;
23         }
24
25         void Poner (T v) {
26             assert (nelementos < 200);
27             datos[nelementos] = v;
28             nelementos++;
29         }
30
31         int size() const {
32             return nelementos;
33         }
34
35         bool Vacia () {
36             return (nelementos == 0);
37         }
38     };

```



### 4.2.2 Celdas enlazadas

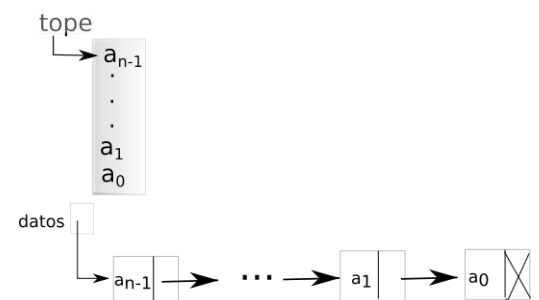
Al ser una pila un contenedor lineal, podemos usar otro de los contenedores lineales vistos para representarlos, en este caso usaremos un conjunto de celdas enlazadas. Las restricciones sobre este conjunto de celdas enlazadas es que solamente se podrá consultar, borrar e insertar por uno de los extremos. Como se puede observar en el siguiente código el *tope* se encuentra en la primera celda. Así la función *Poner* simplemente crea una nueva celda y esta será la primera del conjunto. La función *Quitar* simplemente descarta la primera celda (eliminando su memoria). Y *Tope* devuelve el contenido de la primera celda, finalmente *Vacia* simplemente comprueba si *datos* es 0 o no. Estas funciones

que caracterizan a la Pila: Tope, Poner, Quitar y Vacía se implementan en tiempo constante y sin las desventaja de tener memoria innecesaria como era en el caso de implementar la Pila con un vector estático.

```

1  //Pila.h
2  #include <cassert>
3  template <class T>
4  struct Celda {
5      T d;
6      Celda<T> *sig;
7  };
8
9  template <class T>
10 class Pila {
11     private:
12         Celda<T> *datos;
13         void Copiar (const Pila<T> &p);
14         void Borrar ();
15
16     public:
17         Pila () {datos = 0;}
18         Pila (const Pila<T> &p) {Copiar (p);}
19         ~Pila () {Borrar();}
20         Pila<T> operator= (const Pila<T> &p);
21         T Tope () const;
22         void Quitar ();
23         void Poner (const T &v);
24         bool Vacía () const {return datos==0;}
25         int size()const{
26             int cnt=0;
27             Celda<T> *p=datos;
28             while (p!=0){
29                 p=p->sig;
30                 cnt++;
31             }
32             return cnt;
33         }
34 };

```

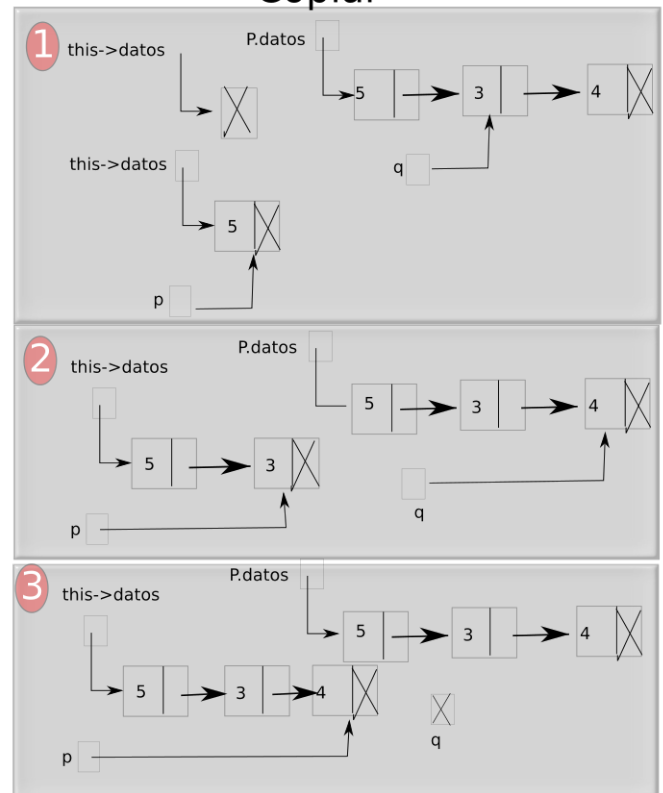


```

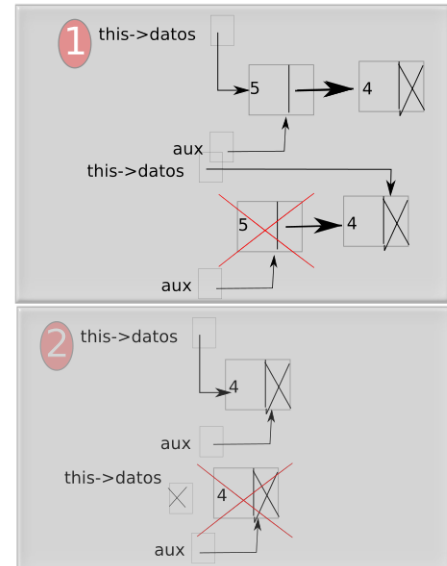
1  //Pila.cpp
2  template <class T>
3  void Pila<T> :: Copiar (const Pila &P) {
4      if (P.datos == 0)
5          datos = 0;
6
7      else {
8          datos = new Celda<T>;
9          datos->d = P.datos->d;
10         Celda<T> *p = datos;
11         Celda<T> *q = P.datos->sig;
12         while (q != 0) {
13             p->sig = new Celda<T>;
14             p = p->sig;
15             p->d = q->d;
16             q = q->sig;
17         }
18         p->sig = 0;
19     }
20 }
21
22 template <class T>
23 void Pila<T> :: Borrar () {
24     while (datos != 0) {
25         Celda<T> *aux = datos;
26         datos = datos->sig;
27         delete aux;
28     }
29 }
30
31 template <class T>
32 Pila<T>& Pila<T> :: operator=(const Pila<T>&P){
33     if (this!=&P){
34         Borrar();
35         Copiar(P);
36     }
37     return *this;
38 }
39
40 template <class T>
41 T Pila<T> :: Tope () const {
42     assert (datos != 0);
43     return datos->d;
44 }
45
46 template <class T>
47 void Pila<T> :: Quitar () {
48     assert (datos != 0);
49     Celda<T> *aux = datos;
50     datos = datos->sig;
51     delete aux;
52 }
53
54 template <class T>
55 void Pila<T> :: Poner (const T &v) {
56     Celda<T> *aux = new Celda<T>;
57     aux->d = v;
58     if (datos == 0) {
59         datos = aux;
60         datos->sig=0;
61     }
62     else {
63         aux->sig=datos;
64         datos=aux;
65     }
66 }

```

## Copiar



## Borrar



### 4.2.3 Vector Dinámico

En este caso la representación la haremos usando un vector dinámico (ver 3.2.2) y el *tope* lo tendremos en el elemento situado en *datos.size()-1*, para un vector con al menos un elemento.

```

1  #include "VD.h"
2  #include <cassert>
3
4  //Pila.h
5  template <class T>
6  class Pila {
7      private:
8          VD<T> datos;
9
10     public:
11         /*Al igual que con la clase conjunto, no necesitamos implementar
12         constructores, destructor ni operador de asignacion ya esta implementado
13         en la clase Vector Dinamico*/
14         T Tope () const {return datos[datos.size()-1];}
15         void Quitar () {datos.Borrar(datos.size()-1);}
16         void Poner (const T &v) {datos.Insertar(v, datos.size());}
17         bool Vacia () const {return datos.size() == 0;}
18     };

```

Eficiencia		
Pila	VD	Celdas
<i>Poner</i>	$O(1)$ <sup>1</sup>	$O(1)$
<i>Quitar</i>	$O(1)$ <sup>1</sup>	$O(1)$
<i>Vacia</i>	$O(1)$	$O(1)$
<i>Tope</i>	$O(1)$	$O(1)$

Con respecto a los tiempos de Poner y Quitar usando para representar a la Pila un vector dinámico es constante ya que estamos considerando el tiempo amortizado, lo que supone un promedio de inserciones y borrados sucesivos. Por otro lado usando las celdas en el peor de los casos todas las funciones tienen un tiempo constante. Si usando un vector dinámico tuviésemos en cuenta el tiempo en el peor de los casos, las funciones Poner y Quitar serían  $O(n)$ .

#### Ejemplo 4.2.1

Obtener la frase invertida de una dada por el usuario.

```

1  #include "Pila.h"
2  #include <iostream>
3  using namespace std;
4  int main(){

```

---

<sup>1</sup>Teniendo en cuenta tiempo amortizado

```
5 Pila<char> mipila;
6 char c;
7 cout<<"Introduce una frase"<<endl; //Pulsa Ctrl+D para terminar
8 while (c=cin.get())
9     mipila.Poner(c);
10 }
11 cout<<"La frase invertida es:";
12 while (!mipila.Vacia()){
13     c=mipila.Tope();
14     mipila.Quitar();
15     cout.put(c);
16
17 }
18 }
```

□

### Ejemplo 4.2.2

Usando una pila convertir un número decimal a un número binario Por ejemplo si el numero que nos da el usuario es 14 el resultado seria 1110.

```
1 #include "Pila.h"
2 #include <iostream>
3 using namespace std;
4 int main(){
5     Pila<int> mipila;
6     int numero;
7     cout<<"Introduce un numero:";
8     cin>>numero;
9     while (numero>0){
10         int digit=(numero %2);
11         mipila.Poner(digit);
12         numero = numero /2;
13     }
14     while (!mipila.Vacia()){
15         int digit = mipila.Tope();
16         mipila.Quitar();
17         cout<<digit;
18     }
19 }
```

□

### Ejercicio 4.1

Suponer que queremos implementar un editor de texto muy básico. El modo de funcionamiento sería escribe el usuario una frase. Si le da a intro el programa saca todo lo escrito hasta el momento por el usuario y espera que le de otra frase para añadirla a las anteriores. Pero si el usuario pulsa la tecla ESC la ultima frase que se introdujo no se tiene en cuenta, se saca por pantalla lo introducido sin esta frase. Este proceso es como implementar la operación Undo. Para implementar este programa usar una Pila de forma que cada entrada conforma lo anterior mas la nueva frase introducida.

□

### Ejemplo 4.2.3

En **notación Polaca o notación Postfijo** las operaciones aritméticas se describen de forma diferente a como las escribimos en notación infijo: (operador izquierda *operador* operando derecha). En la notación Polaca primero van los operandos y detrás los operadores. Por ejemplo si tenemos  $a+b$  en notación Polaca sería  $ab+$ . Otro ejemplo sería si tenemos en infijo  $3 - 4 + 5$  en notación Polaca sería  $3 4 - 5 +$ . Usando una pila podemos evaluar una expresión en notación polaca, suponiendo que los operadores son  $+, -, /, *$  y los operandos son valores enteros. Por ejemplo en el anterior ejemplo con la expresión  $3 4 - 5 +$  tendríamos:

Entrada	Operación	Pila
3	Poner(3)	3
4	Poner(4)	3   4
-	restar	-1
5	Poner(5)	-1   5
+	sumar	4

Ahora veamos el código de este proceso:

```

1  #include "Pila.h"
2  #include <string>
3  #include <sstream>
4  #include <iostream>
5  using namespace std;
6  void QuitarBlancos(string &expresion){
7      while (expresion.size()>0 && expresion[0]==' ')
8          expresion= expresion.substr(1,str::npos);
9  }
10
11
12 bool Operador(string &expresion, char &operador){
13     QuitarBlancos(expresion);
14     if (expresion.size()>0){
15         if (expresion[0]=='+' || expresion[0]=='-' ||
16             expresion[0]=='*' || expresion[0]=='/'){
17             operador = expresion[0];

```



```
18     expresion= expresion.substr(1,str::npos);
19     return true;
20
21 }
22 }
23 return false;
24 }
25
26 void GetOperando(string & expresion,int &operando){
27
28     QuitarBlancos(expresion);
29     if (expresion.size()>0){
30         stringstream ss;
31         string aux;
32
33         ss.str(expresion);
34         ss>>aux; //hasta el primer separador
35         //le quitamos a expresion lo leido en aux
36         expresion=expresion.substr(aux.size(),str::npos);
37         //convertimos de string a int
38         ss.str(aux);
39         ss>>operando;
40     }
41 }
42 int main(){
43     string expresion;
44     cout<<"Introduce una expresion (con separadores espacio en blanco):";
45     cin>>expresion;
46     Pila<int>mipila;
47     while (expresion.size()>0){
48         char operador;
49         if (Operador(expresion,operador)){ // si lo que hay es un operador
50             //sacamos de la pila los dos operandos
51             //y operamos y el resultado se pone en la pila
52             int op2=mipila.Tope();
53             mipila.Quitar();
54             int op1=mipila.Tope();
55             mipila.Quitar();
56             switch(operador){
57                 case '+': int r=op1+op2;
58                     mipila.Poner(r);
59                     break;
60                 case '-': int r=op1-op2;
```

```

61         mipila.Poner(r);
62         break;
63     case '*': int r=op1*op2;
64         mipila.Poner(r);
65         break;
66     case '/': int r=op1/op2;
67         mipila.Poner(r);
68         break;
69 }
70 else {//Operando
71     int operando;
72     GetOperando(expresion,operando);//obtiene de expresion el operando
73     mipila.Poner(operando);
74 }
75 }
76 cout<<"El resultado es: "<<mipila.Tope();
77 }

```

□

### Ejercicio 4.2

Usando el T.D.A Pila escribir un programa que convierta una notación infija a notación postfija o Polaca y viceversa.

□

## 4.3 Cola

1. **Especificación:** Son estructuras de datos lineales que contienen una secuencia de datos

$$\{a_0, a_1, \dots, a_n\}$$

Y están especialmente diseñadas para hacer las inserciones por un extremo y los borrados y consultas por otro. El extremo en el que están el primer elemento ( $a_0, a_1, \dots$ ) se llama *frente*, y es por el que se hacen las consultas y borrados. El extremo en el que están los últimos valores ( $a_n$ ) se llama *última* y es por el que se realizan las inserciones. Las colas responden a la política FIFO (*First Input First Output*).

2. **Operaciones típicas:**

- *Frente* → consulta o accede al elemento en el frente
- *Vacía* → devuelve true si la cola está vacía
- *Quitar* → elimina el elemento que está en el frente
- *Poner* → añade un nuevo elemento por el final (la posición última).

3. **Implementación:**

- Con celdas enlazadas y dos punteros (todas las operaciones nos cuestan  $O(1)$ )
- Con vectores

### 4.3.1 Celdas enlazadas y dos punteros

```

1  //cola.h
2  #ifndef _COLA_H
3  #define _COLA_H
4
5  template <class T>
6  class Cola {
7  private:
8      //Fuera del entorno de la clase, no existe celda
9      //para que exista, lo definimos o bien fuera o en la parte publica
10     struct Celda {
11         T dato;
12         Celda<T> *sig;
13         Celda<T> (const T &d, Celda<T> *s) {
14             dato = d;
15             sig = s;
16         }
17     };
18     Celda *primera, *ultima;
19     void Copiar (const Cola<T> &c);
20     void Borrar ();
21
22 public:
23     Cola () {primera=ultima=0;}
24     Cola (const Cola<T> &c) {Copiar(c);}
25     ~Cola () {Borrar();}
26     Cola<T> & operator= (const Cola<T> &c);
27     void Poner (const T &d);
28     void Quitar ();
29     T Frente ();
30     bool Vacia () const {return primera==0;}
31 };
32 #include "Cola.cpp"
33 #endif

```

```

1  //Cola.cpp
2  template <class T>
3  void Cola<T>::Copiar (const Cola<T> &c) {
4      if (c.primera==0)
5          primera = ultima = 0;
6      else {
7          primera = new Celda<T> (c.primera->dato, 0);
8          ultima = primera;
9          Celda<T> *q = c.primera->sig;

```

```
10         while (q!=0) {
11             ultima->sig = new Celda<T> (q->dato, 0);
12             ultima = ultima->sig;
13             q = q->sig;
14         }
15     }
16 }
17
18 template <class T>
19 void Cola<T>::Borrar() {
20     while (primera != 0) {
21         Celda<T> *aux = primera;
22         primera = primera->sig;
23         delete aux;
24     }
25     ultima = 0;
26 }
27
28 template <class T>
29 Cola<T> & Cola<T>::operator= (const Cola &c) {
30     if (this != &c) {
31         Borrar ();
32         Copiar (c);
33     }
34     return *this;
35 }
36
37 template <class T>
38 void Cola<T>::Poner(const T &e) {
39     Celda<T> *aux = new Celda<T>(e,0);
40     if (primera == 0)
41         primera = ultima = aux;
42     else {
43         ultima->sig = aux;
44         ultima = aux;
45     }
46 }
47
48 template <class T>
49 void Cola<T>::Quitar() {
50     assert (primera != 0);
51     Celda<T> *aux = primera;
52     primera = primera->sig;
```

```

53     delete aux;
54 }
55
56 template <class T>
57 T Cola<T>::Frente() const {
58     assert(primer != 0);
59     return primera->dato;
60 }

```

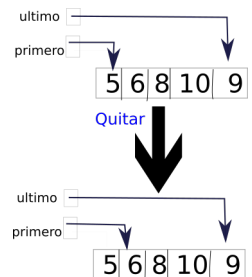
### 4.3.2 Vectores

**Coste de las operaciones** : las operaciones típicas de una cola con una implementación basada en vectores tiene una eficiencia de:

1. *Vacía*  $\rightarrow$  return  $n == 0$ , tiene eficiencia  $O(1)$
2. *Frente*  $\rightarrow$  return  $v[0]$ , tiene eficiencia  $O(1)$
3. *Poner*  $\rightarrow v[n] = \text{nuevo\_elemento}$ ; si tenemos memoria suficiente, tiene eficiencia  $O(1)$  y si tenemos que ampliar el vector,  $O(n)$  Si tenemos en cuenta el *tiempo amortizado* realmente nos costaría en promedio  $O(1)$
4. *Quitar*  $\rightarrow$  debemos hacer un for para desplazar todos los elementos por lo que tenemos eficiencia  $O(n)$

Para mejorar la eficiencia de *Quitar* pensemos en como se accede a los elementos del vector.

Debemos pasar de  $O(n)$  a  $O(1)$  para ello podemos tener dos índices extra: uno que apunte al primer elemento y otro que apunte al último. Así, cuando queramos eliminar un elemento sólo debemos desplazar el primer índice dejando ese elemento como valor basura. El problema de esta solución es que puede llegar un momento en el que se colapse la memoria. Para solucionarlo, usamos *vectores circulares*: cuando nos quedamos sin espacio por el final (el índice del primer elemento y el del último se igualan), seguimos rellenando por el principio.





```
21     ultimo = c.ultimo;
22     n = c.n;
23     datos = new T [capacidad];
24     for (int i=0; i<n; i++)
25         datos[(i+primero)%capacidad]=c.datos[(i+primero)%capacidad];
26 }
27
28 template <class T>
29 Cola<T>::Cola(int tam) {
30     capacidad=tam;
31     primero=ultimo=0;
32     n=0;
33     datos=new T [capacidad];
34 }
35
36 template <class T>
37 Cola<T>::Cola(const Cola<T> &c) {
38     Copiar(c);
39 }
40
41 template <class T>
42 Cola<T>::~~Cola(){
43     if (datos != 0) {
44         delete [] datos;
45     }
46 }
47
48 template <class T>
49 Cola<T> & Cola<T>::operator= (const Cola<T> &c) {
50     if (this != &c) {
51         if (datos != 0)
52             delete [] datos;
53
54         Copiar(c);
55     }
56     return *this;
57 }
58
59
60 template <class T>
61 T Cola<T>::Frente()const {
62     assert(n>0);
63     return datos[0];
```

```

64 }
65
66 template <class T>
67 void Cola<T>::Poner(const T &v) {
68     if (n==capacidad)
69         resize (2*capacidad);
70
71     datos[ultimo]=v;
72     ultimo=(ultimo+1)%capacidad;
73     n++;
74 }
75
76 template <class T>
77 void Cola<T>::Quitar () {
78     primero = (primero+1)%capacidad;
79     n--;
80     if (n<(capacidad/4))
81         resize(capacidad/2);
82 }
83
84 //Ahora, si no hacemos resize tenemos eficiencia O(1)

```

### Ejemplo 4.3.1

Tenemos una cola con enteros y queremos eliminar todos los elementos repetidos **consecutivos** en la cola. Por ejemplo si la cola *c* contiene:

1 1 2 2 2 5 5 1

El resultado para este ejemplo sería :

1 2 5 1

Antes de abordar este problema tenemos que recordar que es imposible acceder a cualquier otro elemento de la cola que no sea el frente, por ello para resolver este ejercicio haremos uso de una cola auxiliar en la que vamos a ir introduciendo los elementos con los que nos quedaremos. Una vez analizados todos los elementos volcaremos los elementos de la cola auxiliar en nuestra cola de entrada.

```

1  #include "Cola.h"
2
3  void EliminarConsecutivos (Cola<int> &c)
4  {
5      Cola<int> caux;
6

```



```

7     while (!c.Vacia())
8     {
9         int d = c.Frente();
10        c.Quitar();
11        caux.Poner(d);
12        while (!c.Vacia() && d==c.Frente())
13            c.Quitar();
14    }
15
16    while (!caux.Vacia())
17    {
18        int d = caux.Frente();
19        c.Poner(d);
20        caux.Quitar();
21    }
22 }

```

□

### Ejemplo 4.3.2

Representa el T.D.A. Cola haciendo uso de T.D.A. Pila.

Para ello representaremos nuestra cola con dos pilas. Una primera pila sobre la que hago las consultas y borrados  $P_2$  y otra para realizar las inserciones  $P_1$ . Vamos a ver, antes de la implementación, varios ejemplos de cómo funciona una cola con dos pilas.

1. **Insertar:** 3 2 1 9

P1: 3 2 1 9      donde 9 es el tope de la pila

P2:

2. **Consultar frente:** para ello debemos insertar los elementos en P2 para obtener el primer elemento que se insertó en la cola:

P1:

P2: 9 1 2 3      en este caso, el frente sería 3

3. **Insertar:** 5 7

P1: 5 7

P2: 9 1 2 3

4. **Quitar (el frente):**

P1: 5 7

P2: 9 1 2

Podemos quitar elementos de P2 hasta dejarla vacía, y si quisiésemos seguir quitando, insertaríamos los elementos de P1 en P2 y haríamos la operación de quitar el frente.

```

1  #include "Pila.h"
2
3  class Cola
4  {

```

```

5  private:
6      Pila<int> p1; //insertar
7      Pila<int> p2; //frente y quitar
8
9  public:
10     int Frente () {
11         if (p2.Vacia()) {
12             while (!p1.Vacia()) {
13                 int d = p1.Tope();
14                 p2.Poner(d);
15                 p1.Quitar();
16             }
17         }
18         return p2.Tope();
19     }
20
21     bool Vacia() const {
22         return p1.Vacia() && p2.Vacia();
23     }
24
25     void Poner (int d) {
26         p1.Poner(d);
27     }
28
29     void Quitar () {
30         if (p2.Vacia()) {
31             while (!p1.Vacia()) {
32                 int d = p1.Tope();
33                 p2.Poner(d);
34                 p1.Quitar();
35             }
36         }
37         p2.Quitar();
38     }
39 };

```

□

**Ejercicio 4.3**

Representar el T.D.A. Pila usando el T.D.A Cola. Establecer cuál es la eficiencia de las operaciones Vacía, Tope, Poner y Quitar.

□

## 4.4 Cola con prioridad

**Especificación.** Contienen una secuencia de valores especialmente diseñadas para realizar accesos y borrados por el frente. Pero a diferencia de las colas normales, las inserciones se realizan en cualquier punto de acuerdo a un criterio de prioridad. Cada dato que se guarda en la cola con prioridad debe componerse del dato y su prioridad.

Las operaciones típicas son:

1. **Frente:** devuelve el elemento en el frente
2. **Prioridad:** devuelve la prioridad del elemento en el frente
3. **Quitar:** elimina el elemento que está en el frente. Este es el más prioritario.
4. **Vacía:** indica si la cola está vacía
5. **Poner:** inserta un nuevo elemento de acuerdo a su prioridad

Para representar una cola con prioridad al igual que hicimos con las colas simples podríamos plantearnos diferentes estructuras de datos: vector dinámico, vectores circulares, celdas enlazadas. Debido a la eficiencia que resulta en las operaciones vamos a realizar la representación usando un conjunto de celdas enlazadas.

### 4.4.1 Representación con celdas enlazadas

Como se puede ver en el siguiente código se define la estructura *info* que será el tipo que almacena la cola con prioridad.

```
1  template <class Tprio, class T>
2  struct info {
3      Tprio prio;
4      T elemento;
5  };
```

El campo *prio* será la prioridad del dato (que se almacena en el campo *elemento*). Ambos campos son templates, en este caso hemos usado dos templates para que el usuario tenga la libertad de definir la prioridad con un tipo diferente al tipo del dato. Por ejemplo se podría hacer las siguiente instanciaciones:

```
1  ...
2  info<int,int> a; //la prioridad int y el elemento int
3  info<char,int> b; //la prioridad char y el elemento int
4  ...
```

El único detalle para poder usar la cola con prioridad instanciando la prioridad a un tipo concreto es que el usuario deberá ser consciente de que para establecer la prioridad el tipo al que se instancie *Tprio* debe tener definido el operador relacional menor. Veamos como sería la implementación de la cola con prioridad.

```
1  template <class Tprio, class T>
2  struct Celda {
3      info<Tprio, T> dato;
```

```

4      Celda<Tprio, T> *sig;
5  };
6
7  template <class Tprio, class T>
8  class ColaPrio {
9  private:
10     Celda<Tprio, T> *primera;
11     void Copiar (const ColaPrio<Tprio, T> &p);
12     void Borrar ();
13
14 public:
15     ColaPrio() {primera=0;}
16     ColaPrio(const ColaPrio<Tprio, T> &c);
17     ~ColaPrio();
18     ColaPrio<Tprio, T> & operator=(const ColaPrio<Tprio, T> &cp);
19     T Frente() const;
20     Tprio Prioridad_Frente() const;
21     void Poner(const Tprio &tp, const T &elemento);
22     void Quitar ();
23     void size() const;
24     bool Vacia() const {primera==0;}
25 };
26 #include "ColaPrio.cpp"
27
28 /*La mayoría de operaciones son igual que en las colas, vamos a estudiar
29 las que se diferencian de las operaciones de las colas*/
30
31 //ColaPrio.cpp
32 template <class Tprio, class T>
33 T Cola_Prio<Tprio,T>::Frente()const{
34     assert(primera!=0); //nos aseguramos de que la cola no esta vacia
35     return primera->dato.elemento;
36 }
37
38 template <class Tprio, class T>
39 Tprio Cola_Prio<Tprio,T>::Prioridad_Frente() const {
40     assert(primera!=0);
41     return primera->dato.prio;
42 }
43
44 template <class Tprio, class T>
45 void Cola_Prio<Tprio,T>::Poner(const Tprio &tp, const T &e) {
46     //Creamos una nueva celda para guardar los datos que tenemos
47     Celda<Tprio,T> *aux = new Celda<Tprio, T>;

```

```

18     aux->dato.elemento = e;
19     aux->dato.prio = tp;
20
21     //Primer caso: la cola esta vacia
22     if (primera == 0) {
23         primera = aux;
24         primera->sig = 0;
25     }
26
27     //Segundo caso: la prioridad es mayor a la prioridad del frente
28     else {
29         if (primera->dato.prio < tp) {
30             aux->sig = primera;
31             primera = aux;
32         }
33
34         //Tercer caso: la celda se inserta en cualquier parte de la cola
35         else {
36             Celda<Tprio,T> *p = primera;
37
38             //buscamos la celda en la que insertar la nuestra
39             while (p->sig->dato.prio > tp && p->sig!=0)
40                 p=p->sig;
41
42             //Una vez la encontramos insertamos la nuestra ahi
43             aux->sig = p->sig;
44             p->sig=aux;
45         }
46     }
47 }
48
49 template <class Tprio, class T>
50 void Cola_Prio <Tprio,T>::Quitar() {
51     assert(primera!=0);
52     Celda<Tprio,T> *aux=primera;
53     primera=primera->sig;
54     delete aux;
55 }

```

Como se puede observar la eficiencia de las funciones son constantes a excepción de *Poner* que realiza una búsqueda secuencial para establecer donde realizar la inserción. Usando una representación en la que tuviésemos acceso directo a cada elemento, como por ejemplo en un vector dinámico, podríamos realizar una búsqueda binaria y así obteniendo una eficiencia en el peor de los casos de *Poner* de  $O(\log_2(n))$ .

## 4.5 Lista

Es un tipo de dato lineal que contiene una secuencia de elementos  $\{a_0, a_1, \dots, a_{n-1}\}$ . Está especialmente diseñado para realizar inserciones, borrados y consultas desde cualquier posición.

Sus operaciones típicas son:

1. *Set*  $\rightarrow$  modifica el elemento en una posición
2. *Get*  $\rightarrow$  devuelve el elemento en una posición
3. *Borrar*  $\rightarrow$  borra un elemento de una determinada posición
4. *Insertar*  $\rightarrow$  inserta un elemento en una determinada posición
5. *size*  $\rightarrow$  devuelve cuántos elementos hay en la lista

Su interfaz es:

```

1  #ifndef __LISTA_H
2  #define __LISTA_H
3
4  template <class T>
5  class Lista {
6      private:
7          //Implementacion elegida
8
9      public:
10         Lista();
11         Lista(const Lista &l);
12         ~Lista();
13         Lista & operator=(const Lista<T> &l);
14         T get (int posicion) const;
15         void set(int posicion, const T &e);
16         int size() const;
17         void Insertar (int posicion, const T &e);
18         void Borrar (int posicion);
19     };
20 #endif

```

Como se puede ver, en las funciones que caracterizan a la lista existe el concepto *posicion*. Si representamos la clase Lista como un vector, tener la posición como int está bien, pero si elegimos celdas enlazadas, cada vez que accedamos a un elemento nos costará  $O(n)$ . En vez de eso, sería más interesante hacer que posición fuera un puntero al elemento a acceder. De esta forma crearemos un nuevo tipo de dato *Posición*, que direcciona un elemento de una lista. Además este nuevo tipo de dato, *Posición*, tendrá las funcionalidades de: pasar el siguiente elemento de la lista, retroceder al anterior de la lista, obtener a partir del objeto *posicion* el valor del elemento al que apunta en la lista, compararse con otra posición.

Al final del código hemos puesto que la clase Lista es amiga de Posición. Esto es así ya que la clase Lista deberá acceder a la representación de la clase Posición, en particular cuando la clase lista quiera devolver una determinada Posición. Dos posiciones relevantes de la lista serán: la posición comienzo (begin) y la posición fin (end). La forma de implementar las funciones de Posición esta determinado

por la representación que hagamos del contenedor (Lista) que recorre. La implementación que hemos hecho de *Posición* es válida para una implementación de lista como un vector dinámico como se muestra a continuación.

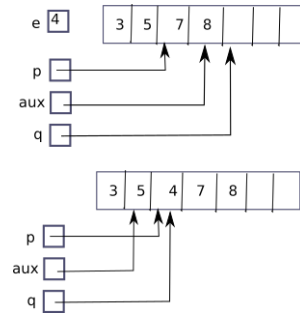
#### 4.5.1 Implementación con vectores

```
1  template <class T>
2  class Lista {
3      private:
4          T *datos;
5          int n;
6          int reservados;
7          void resize(int tam);
8          void Copiar(const Lista<T> &l);
9
10     public:
11         Lista (int tam = 0);
12         Lista (const Lista<T> &l);
13         ~Lista();
14         Lista<T> & operator= (const Lista<T> &l);
15         T Get(Posicion<T> p) const {return *p.i;}
16         void Set(Posicion<T> p, const T &e);
17         int size() const;
18         void Insertar(Posicion<T> p, const T &e);
19         void Borrar(Posicion<T> p);
20         //En particular por estas dos funciones begin y end la clase Lista
21         // es amiga de Posicion
22         Posicion begin() const; //devuelve una posicion al inicio de la lista
23         Posicion end() const; //devuele una posicion al final de la lista
24     };
25 #endif
```

```

1  //Lista.cpp
2  template <class T>
3  void Lista::Set(Posicion<T> p, const T&e) {
4      assert(p.i!=0);
5      *(p.i)=e;
6  }
7
8  template <class T>
9  Posicion<T> Lista<T>::Insertar (Posicion<T> p, const T&e) {
10     if (n == reservados/2)
11         resize (2*reservados);
12
13     Posicion<T> q = end(), aux;
14     aux = q;
15     //posicion anterior a la ultima
16     --aux;
17
18     for (; q != p; q--, aux--)
19         *q = *aux;
20
21     *q=e;
22     n++;
23     return q;
24 }
25
26 template <class T>
27 Posicion<T> Lista<T>::Borrar(Posicion<T> p) {
28     Posicion<T> siguiente = p;
29     siguiente++;
30
31     for (Posicion<T> q = p; siguiente != end(); q++, siguiente++)
32         *q = *siguiente;
33
34     n--;
35
36     if (n < (reservados/4))
37         resize(reservados/2);
38
39     return p;
40 }
41
42 Lista::Posicion Lista::begin()const {
43     Posicion p;
44     p.i = &(datos[0]);
45     return p;
46 }
47
48
49 Lista::Posicion Lista::end()const {
50     Posicion p;
51     p.i=&(datos[n]);
52     return p;
53 }

```





Así, si por ejemplo queremos mostrar los elementos de una lista, lo único que tendríamos que hacer sería:

```
1  for (Posicion p=L.begin(); p!=L.end(); ++p)
2      cout << *p;
```



Código  
válido para  
cualquier  
representación de  
Lista y Posición

En este código se crea un objeto de tipo Posición  $p$ . Este objeto se inicializa al comienzo de la lista usando el método  $L.begin()$  que lo que hace internamente es que  $p.i = \&(datos[0])$  (también hubiese sido válido  $p.i = datos$ ). Mientras que  $p$  no apunte al final de la lista ( $L.end()$ ) se saca por la salida estándar el elemento al que apunta  $p$ , usando para ello el operador  $*$  (es decir  $*(p.i)$ ). A continuación  $p$  pasa a apuntar al siguiente elemento de la lista y se vuelve a comprobar si es diferente del fin de la lista. Como se puede observar al usar la interfaz de Posición y usando las funciones  $begin$  y  $end$  de Lista este código será válido para cualquier representación que hagamos de Lista y de Posición, siendo esta última la encargada de saber como pasar al siguiente elemento de la lista y como acceder al elemento.

#### 4.5.2 Implementación con celdas enlazadas

En este caso, la clase Posicion cambia un poco, ya que el operador  $++$  y  $--$  no se pueden implementar de igual manera. Ahora para situarnos en la siguiente posición usaremos la dirección siguiente de la Celda (ya que la memoria no es consecutiva). Igualmente para poder implementar el operador  $--$  tendremos que partir del inicio de la lista y ver cuando el siguiente es el actual para quedarnos en el anterior.

Para simplificar la notación, fijaremos el tipo del elemento que almacena la lista a carácter.

```
1  #ifndef __LISTA_H
2  #define __LISTA_H
3  typedef char Tbase
4
5  struct Celda {
6      Tbase ele;
7      Celda *sig;
8  }
9
10 class Posicion {
11     private:
12         Celda *punt;           //apunta a la posicion que le digamos
13         Celda *primera;        //apunta a la primera celda de la lista,
14                                //lo necesitamos para el operador --
15     public:
16         Posicion & operator ++ () {
17             punt = punt->sig;
18             return *this;
19         }
20 }
```

```

21     Posicion & operator -- () {
22         //Caso1: si posicion es begin
23         if (punt==primera){
24             punt =0;
25             return *this;
26         }
27         //Caso2: es cualquier otra posicion
28         //Buscamos la anterior a nuestra celda con el
29         //puntero primera
30
31         Celda *aux = primera;
32
33         while (aux != 0 && aux->sig != punt)
34             aux++;
35
36         punt = aux;
37
38         return *this;
39     }
40
41     Tbase & operator * () {
42         return punt->ele;
43     }
44
45     //El resto de funciones no cambian
46
47     friend class Lista;
48 };

```

Usando esta implementación de la clase Posicion, nuestra Lista con celdas enlazadas sería:

```

1  //Lista.h
2  class Lista {
3      private:
4          Celda *primera;
5          void Copiar (const Lista &l);
6          void Borrar_All();
7
8      public:
9          Lista() {primera = 0;}
10         Lista(const Lista &l) {Copiar(l);}
11         ~Lista(){Borrar_All();}
12         Lista & operator=(const Lista &l);
13         void Insertar(Posicion p, Tbase e);

```

```
14         void Borrar(Posicion p);
15         Tbase Get(Posicion p) const {return *p;}
16         void Set(Posicion p, Tbase v) {(*p)=v;}
17         int size() const;
18         Posicion begin() const;
19         Posicion end() const;
20     };
21 #endif

1 //Lista.cpp
2 #include "Lista.h"
3
4 void Lista::Copiar(const Lista &l) {
5     //Primer caso: lista vacia
6     if (l.primeras == 0)
7         primeras = 0;
8
9     else {
10         //Al menos hay una celda en la lista
11         primeras = new Celda;
12         primeras->ele = l.primeras->ele;
13
14         Celda *p = primeras, *q = l.primeras->sig;
15
16         while (q != 0) {
17             p->sig = new Celda;
18             p = p->sig;
19             p->ele = q->ele;
20             q = q->sig;
21         }
22         p->sig=0;
23     }
24 }
25
26 void Lista::Borrar_All() {
27     while (primeras != 0) {
28         Celda *aux = primeras;
29         primeras = primeras->sig;
30         delete aux;
31     }
32 }
33
34 Lista & Lista::operator=(const Lista &l) {
35     if (this != &l) {
```

```
36         Borrar_All();
37         Copiar(l);
38     }
39     return *this;
40 }
41
42 void Lista::Insertar(Posicion p, Tbase e) {
43     Celda *aux = new Celda;
44     aux->ele=e;
45     if (p==begin()) {
46         aux->sig=primera;
47         primera=aux;
48     }
49
50     else {
51         Posicion q=p;
52         --q;
53         aux->sig=p.punt;
54         q.punt->sig=aux;
55     }
56 }
57
58 void Lista::Borrar(Posicion p) {
59     if (p==begin()) {
60         primera = p.punt->sig;
61         delete p.punt;
62     }
63
64     else {
65         Posicion q = p;
66         --q;
67         q.punt->sig = p.punt->sig;
68         delete p.punt;
69     }
70 }
71
72 int Lista::Size()const {
73     int contador=0;
74     Celda *aux=primera;
75
76     while (aux!=0) {
77         contador++;
78         aux=aux->sig;
```

```

79     }
80
81     return contador;
82 }
83
84 Posicion Lista::begin() const {
85     Posicion p;
86     p.punt=primera;
87     p.primera=primera;
88     return p;
89 }
90
91 Posicion Lista::end() const {
92     Posicion p;
93     p.punt=0;
94     p.primera=primera;
95     return p;
96 }

```

Con esta representación tenemos varios problemas:

1. Tenemos dos punteros para representar una posición
2. Gasto en eficiencia del operador — de Posición,  $O(n)$

A continuación vamos a estudiar modificaciones en la representación de Posición para mejorar la eficiencia del operador — y usar el menor numero de punteros posibles.

### 4.5.3 Implementación con celdas enlazadas con cabecera

La cabecera es una celda vacía que se coloca al principio de la lista, la función `begin()` nos devuelve un puntero a esta Celda y la clase `Posicion` siempre direcciona al valor anterior al que realmente apunta.

Como se puede observar en la figura 4.1 la posición internamente direcciona al anterior elemento al que realmente devolvería si aplicamos sobre una posición el operador `*`. Así por ejemplo  $p_1$  se inicia a `begin`. Internamente sus campos `punt` y `primera` apunta a la misma dirección que `cab`, es decir la celda cabecera. Pero cuando se aplica el operador `*`, es decir  $*p_1$  devolverá el objeto con valor 4 (el contenido de la celda siguiente). De la misma forma  $p_2$  se inicia a `end()`. Aunque  $p_2.punt$  apunta a la última celda válida, realmente está apuntando a una posición que no contiene datos válidos, ya que  $p_2.punt \rightarrow sig = 0$ . Un detalle de la siguiente implementación es que posición se ha encapsulado dentro de la clase `Lista`. Así si se quiere declarar un objeto de tipo Posición sería:

```

1  Lista::Posicion p;
2  ...

```

La implementación de la clase `Lista` y `Lista::Posicion` se detalla a continuación:

```

1  //Lista.h
2  #ifndef __LISTA_H
3  #define __LISTA_H

```

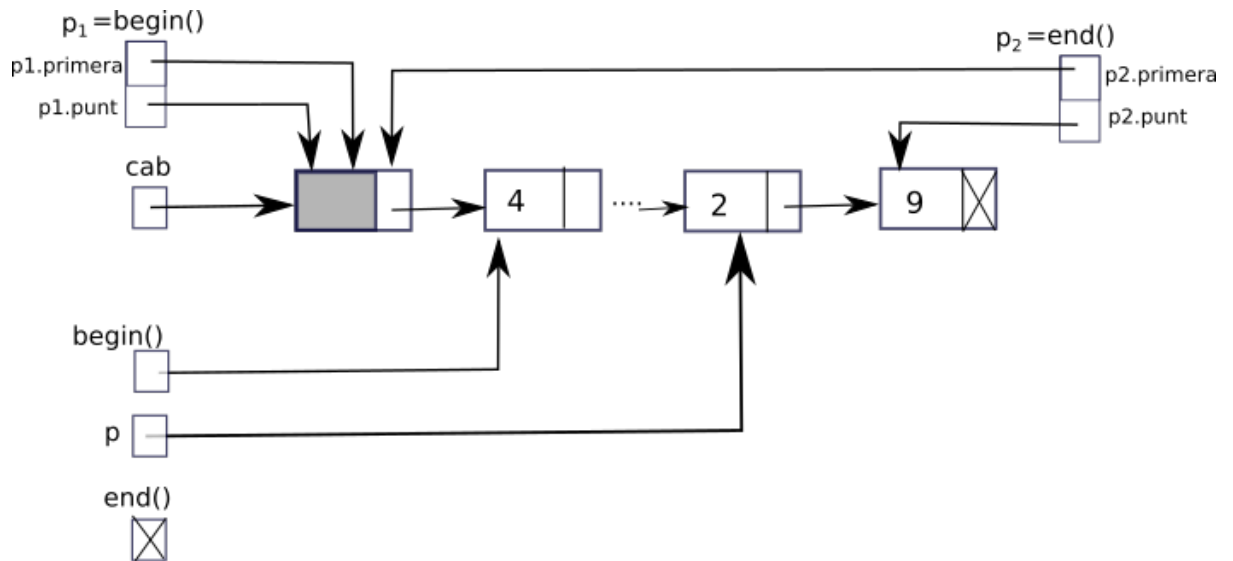


Figura 4.1: Ejemplo de una lista con cabecera. Diferentes posiciones en la lista y como internamente se estructuran estas posiciones

```

4
5 typedef char Tbase;
6 struct Celda
7 {
8     Tbase ele;
9     Celda *sig;
10 };
11
12 class Lista
13 {
14     class Posicion
15     {
16     private:
17         Celda *punt;
18         Celda *primera;
19
20     public:
21         Posicion(): punt(0), primera(0) {}
22         bool operator== (const Posicion &op) {
23             return punt==op.punt;
24         }
25         bool operator!= (const Posicion &op) {
26             return punt!=op.punt;
27         }

```

```

28         Posicion & operator++(){
29             assert(punt!=0);
30             punt=punt->sig;
31             return *this;
32     }
33     Posicion & operator--(){
34         if (primera==punt)
35             punt=0;
36
37         else
38         {
39             Celda *q = primera;
40             while (q->sig!=punt)
41                 q=q->sig;
42             punt=q;
43             return *this;
44         }
45     }
46     Tbase & operator*() {
47         return punt->sig->ele;
48     }
49     //para poder implementar las operaciones
50     //begin y end de Lista tenemos
51     //que hacer amiga Lista de Posicion
52     friend class Lista;
53 };
54
55 private:
56     Celda *cab;
57     void Copiar (const Lista &l);
58     void Borrar_All();
59
60 public:
61     Lista();
62     Lista(const Lista &l);
63     ~Lista();
64     Lista & operator=(const Lista &l);
65     void Insertar (Posicion p, Tbase v);
66     void Borrar (Posicion p);
67     Tbase Get(Posicion p)const;
68     void Set(Posicion p, Tbase v);
69     int size()const;
70     Posicion begin()const;

```

```
71     Posicion end()const;
72 };
73
74 #endif
75
1 //Lista.cpp
2 void Lista::Copiar(const Lista &l)
3 {
4     if (l.cab->sig==0)
5     {
6         cab=new Celda;
7         cab->sig=0;
8     }
9
10    else
11    {
12        cab = new Celda;
13        Celda *p=cab, *q=l.cab;
14        while (q->sig != 0)
15        {
16            p->sig=new Celda;
17            p->sig->ele = q->sig->ele;
18            p=p->sig;
19            q=q->sig;
20        }
21        p->sig=0;
22    }
23 }
24
25 void Lista::Borrar_All()
26 {
27     while (cab->sig!=0)
28     {
29         Celda *aux=cab->sig;
30         cab->sig = cab->sig->sig;
31         delete aux;
32     }
33     delete cab;
34 }
35
36 Lista::Lista()
37 {
38     cab=new Celda;
39     cab->sig=0;
```



```
40 }
41
42 Lista::Lista(const Lista &l)
43 {
44     Copiar(l);
45 }
46
47 Lista::~~Lista()
48 {
49     Borrar_All();
50 }
51
52 Lista & Lista::operator=(const Lista &l)
53 {
54     if (this != &l)
55     {
56         Borrar_All();
57         Copiar(l);
58     }
59
60     return *this;
61 }
62
63 void Lista::Insertar(Posicion p, Tbase v)
64 {
65     Celda *aux=p.punt->sig;
66     p.punt->sig=new Celda;
67     p.punt->sig->ele=v;
68     p.punt->sig->sig=aux;
69 }
70
71 void Lista::Borrar(Posicion p)
72 {
73     assert(p.punt->sig != 0);
74     Celda *aux = p.punt->sig;
75     p.punt->sig=p.punt->sig->sig;
76     delete aux;
77 }
78
79 Posicion Lista::begin()const
80 {
81     Posicion p;
82     p.punt=cab;
```

```

83     p.primer=primera;
84     return p;
85 }
86
87 Posicion Lista::end()const
88 {
89     Posicion p;
90     Celda *aux = cab;
91     while (aux->sig!=0)
92         aux=aux->sig;
93
94     p.punt=aux;
95     p.primer=cab;
96     return p;
97 }

```

Con esta representación de *Posición* hemos mejorado la implementación de los métodos Insertar y Borrar de Lista. Pero seguimos teniendo dos punteros para representar una posición, ya que lo seguimos necesitando para aplicar el operador  $--$ , aunque ahora  $p$  apunte a la anterior. Además desde un punto de vista conceptual físicamente una posición apunta a una dirección diferente a la que objetivamente se usa (la siguiente). Para intentar de nuevo evitar obtener una eficiencia  $O(n)$  en el operador  $--$  veamos la representación usando celdas doblemente enlazadas

#### 4.5.4 Implementación con celdas doblemente enlazadas

El esquema en este caso sería el que se muestra en la figura 4.2. Ahora un objeto de tipo posición direcciona a la celda que realmente apunta. Por lo tanto hemos eliminado ese aspecto ambiguo de la anterior representación. Las celdas por lo tanto tienen dos punteros: uno que apunta a la celda anterior y otro que apunta a la celda siguiente. Como se puede observar el anterior a la primera es 0 al igual que la siguiente a la última.

```

1  struct Celda
2  {
3      char ele;
4      Celda *sig;
5      Celda *ant;
6  };
7
8  class Lista
9  {
10     Class Posicion
11     {
12     private:
13         Celda *punt;
14         Celda *primera;

```

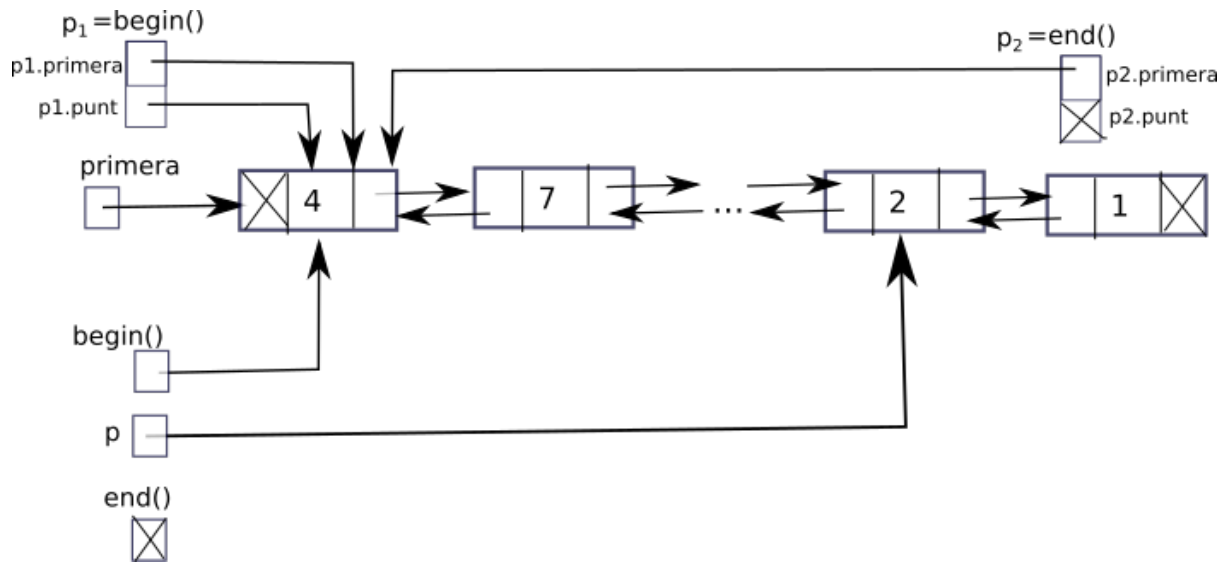


Figura 4.2: Ejemplo de una lista con celdas doblemente enlazadas. Diferentes posiciones en la lista y como internamente se estructuran estas posiciones

```

15
16     public:
17         Posicion(): punt(0), primera(0) {}
18         bool operator== (const Posicion &op) {
19             return punt==op.punt;
20         }
21         bool operator!= (const Posicion &op) {
22             return punt!=op.punt;
23         }
24         Posicion & operator++(){
25             assert(punt!=0);
26             punt=punt->sig;
27             return *this;
28         }
29         Posicion & operator--(){
30             if (primera==punt)
31                 punt=0;
32
33             else
34             {
35                 if (punt==0){//es la ultima
36                     Celda *q=primera;
37                     while (q->sig!=0)
38                         q=q->sig;

```

```

39         punt=q;
40     }
41     else{
42
43         punt=punt->ant;
44
45     }
46 }
47 return *this;
48
49 }
50 Tbase & operator*() {
51     return punt->ele;
52 }
53 friend class Lista;
54 };
55
56 private:
57     Celda *primera;
58     void Copiar (const Lista &l);
59     void Borrar_All();
60
61 public:
62     Lista();
63     Lista(const Lista &l);
64     ~Lista();
65     Lista & operator=(const Lista &l);
66     void Insertar (Posicion p, Tbase v);
67     void Borrar (Posicion p);
68     Tbase Get(Posicion p)const;
69     void Set(Posicion p, Tbase v);
70     int size()const;
71     Posicion begin()const;
72     Posicion end()const;
73 };
74
75 #endif

```

Con esta nueva representación de *Posicion* veamos como cambian las funciones Insertar y Borrar.

```

1 void Lista::Insertar(Posicion p, char e)
2 {
3     Celda *aux=new Celda;
4     aux->ele = e;

```

```

5     if (p==begin())//insertar al principio
6     {
7         aux->sig = primera;
8         if(primer!=0)
9         {
10            aux->ant=0;
11            primera=aux;
12        }
13    }
14    else
15    {
16        aux->sig=p.punt;
17        Posicion q=p;
18        --q;
19        q.punt->sig=aux;
20        aux->ant=q.punt;
21        if (p.punt != 0)
22            p.punt->ant=aux;
23    }
24 }
25
26 void Lista::Borrar(Posicion p)
27 {
28     Celda *aux=p.punt;
29     if (p==begin())
30     {
31         primera=primera->sig;
32         if (primera!=0)
33             primera->ant=0;
34         delete aux;
35     }
36     else
37     {
38         p.punt->ant->sig=p.punt->sig;
39         if (p.punt->sig!=0)
40             p.punt->sig->ant=p.punt->ant;
41         delete aux;
42     }
43 }

```

El problema es que seguimos necesitando representar la clase *Posicion* con dos punteros. Aún el operador `--` en el peor de los casos nos sigue costando  $O(n)$ . Esto ocurre cuando queremos aplicar el operador `--` sobre la posición `end()`. Como la posición `end()` tiene un puntero apuntando 0 no sabemos quién es la celda anterior. Por lo tanto tenemos que partir de la primera hasta llegar a una celda cuyo

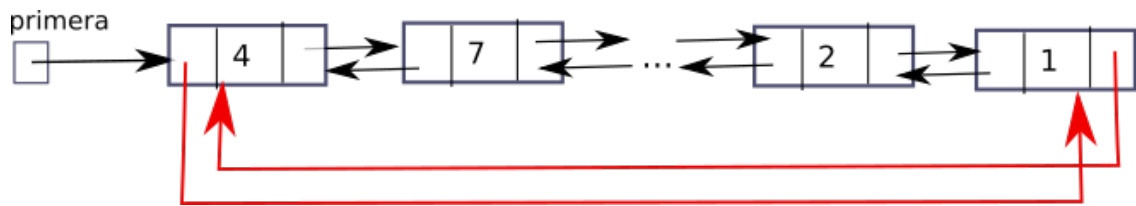


Figura 4.3: Ejemplo de una lista circular

siguiente sea 0. En cualquier otro caso el coste del operador — es  $O(1)$ . En nuestra búsqueda para solucionar el problema de tener dos punteros y bajar el tiempo en el peor de los casos del operador — seguimos refinando nuestra implementación. Con este fin vamos a analizar la implementación usando listas circulares.

#### 4.5.5 Listas circulares

Con esta implementación mantenemos en la celda un puntero a la celda siguiente y anterior. La única diferencia con la anterior representación es que la celda siguiente a la última es la primera, y la anterior a la primera es la última. De forma que detectar la última celda consiste en establecer si la siguiente a ella es la primera. Con esta representación solucionamos la eficiencia en el peor de los casos del operador —. El problema con esta representación es que `begin` y `end` devuelve la misma posición. Y por lo tanto no se distinguen. Así que la siguiente solución que vamos a ver son celdas doblemente enlazadas circulares con cabecera.

#### 4.5.6 Listas como celdas doblemente enlazadas circulares con cabecera

En la figura 4.4 se muestra un ejemplo de una lista circular con cabecera y doblemente enlazada. Además se muestra las posiciones `begin` y `end` donde se sitúan.

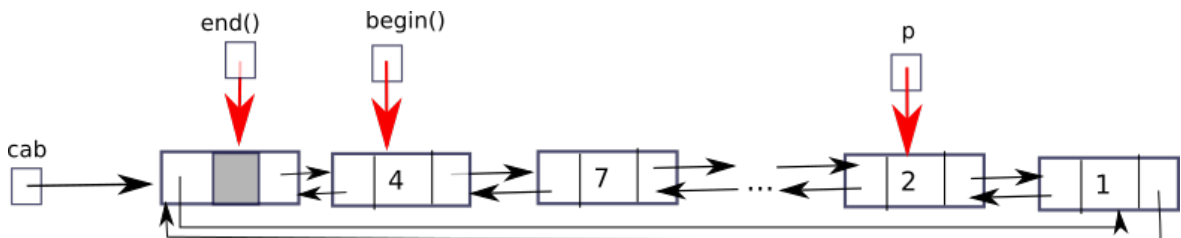


Figura 4.4: Ejemplo de una lista circular con celdas doblemente enlazadas y con cabecera

Con esta representación, con cabecera, `end()` apuntará a la celda cabecera distinguiéndola así de la posición de comienzo, que es la primera celda con un dato válido. Ahora implementar el operador — simplemente hay que seguir la dirección del campo anterior de la celda sin distinciones. Y además posición se representará con un único puntero. Al igual que en la implementación de listas con celdas doblemente enlazadas nuestra celda tendrá dos punteros uno al siguiente y otro al anterior

```
1 typedef char Tbase;
2 struct Celda{
```

```

3   Tbase ele;
4   Celda * ant;
5   Celda * sig;
6   };

```

Ahora tenemos la interfaz de Lista y describimos dentro también la clase Posición que tiene un único puntero.

```

1  class Lista{
2      private:
3          Celda *cab;
4
5      public:
6          Lista();
7          Lista(const Lista &l);
8          ~Lista();
9          Posicion Insertar(Posicion p, Tbase e);
10         Posicion Borrar(Posicion p);
11         int size()const;
12         class Posicion{
13             private:
14                 Celda *punt;
15             public:
16                 Posicion(): punt(0){}
17                 Posicion & operator ++(){
18                     punt=punt->sig;
19                     return *this;
20                 }
21                 Posicion &operator--(){
22                     punt=punt->ant;
23                     return *this;
24                 }
25                 Tbase & operator *(){
26                     return punt->ele;
27                 }
28                 bool operator==(const Posicion &p)const{
29                     return punt==p.punt;
30                 }
31                 bool operator!=(const Posicion &p)const{
32                     return punt!=p.punt;
33                 }
34                 friend class Lista;
35         };//fin de la clase Posicion
36

```

```

37     //begin y end
38     Posicion begin()const{
39         Posicion p;
40         p.punt=cab->sig;
41         return p;
42     }
43     Posicion end()const{
44         Posicion p;
45         p.punt=cab;
46         return p;
47     }
48 }; //fin de lista

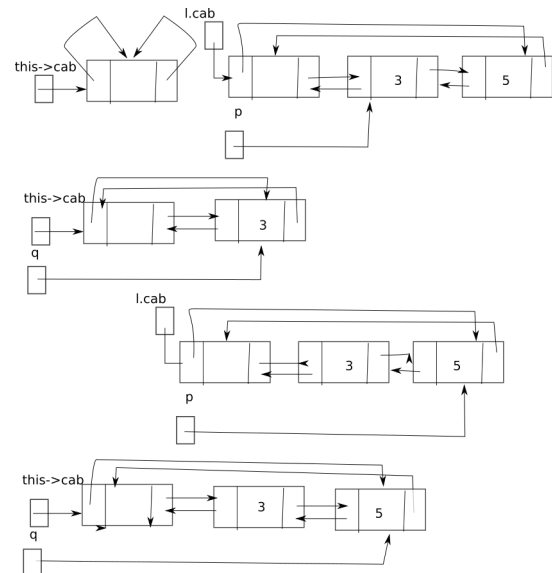
```

A continuación detallaremos la implementación de Lista

```

1  Lista::Lista(){
2      cab = new Celda;
3      cab->sig=cab;
4      cab->anterior=cab;
5  }
6  Lista::Lista(const Lista &l) {
7      cab = new Celda;
8      cab->sig=cab;
9      cab->ant=cab;
10     //Empezamos a copiar
11     Celda *p = l.cab->sig;
12     //mientras no demos la vuelta
13     while (p!=l.cab){
14         Celda *q=new Celda;
15         q->ele=p->ele;
16         q->ant=cab->ant;
17         cab->ant->sig=q;
18         cab->ant=q;
19         cab->sig=cab;
20         p=p->sig;
21     }
22 }

```



```

23 int Lista::size(){
24     Posicion p=begin();
25     int cnt=0;
26     for (;p!=end();++p,++cnt);

```



```

27     return cnt;
28 }
29
30 Lista::~Lista(){
31     while (begin()!=end())
32         borrar(begin());
33 }
34 delete cab;
35 }
36 Lista & Lista::operator=(const Lista &l){
37     Lista aux(l);
38     Celda *p;
39     p=this->cab;
40     this->cab=aux.cab;
41     aux.cab=p;
42     return *this;
43 }
44 Posicion Lista::Insertar(Posicion p, Tbase e){
45     Celda *q=new Celda;
46     q->ant=p.punt->ant;
47     q->sig=p.punt;
48     p.punt->ant=q;
49     q->ele=e;
50     p.punt=q;
51     return p;
52 }
53 //p debe ser una posicion diferente a end
54 Posicion Lista::Borrar(Posicion p){
55     Celda *q=p.punt;
56     q->ant->sig=q->sig;
57     q->sig->ant=q->ant;
58     p.punt=q->sig;
59     delete q;
60     return p;
61 }
62 }

```

### Ejemplo 4.5.1

Dado el T.D.A. Lista (suponiendo que se ha implementado con templates) con el T.D.A. Posicion asociado, Implementar el T.D.A. Pila de caracteres.

```

1  class Pila{
2      private:

```

```

3      Lista<char> datos;
4  public:
5      char Tope() const{
6          *(datos.begin());
7      }
8      void Poner(char v){
9          datos.Insertar(datos.begin(),v);
10     }
11     void Quitar(){
12         datos.Borrar(datos.begin());
13     }
14     bool Vacia() const{
15         return datos.size()==0;
16     }
17 }

```

□

**Ejemplo 4.5.2**

Construir una función que a partir de una lista de enteros obtenga una nueva lista con los datos de la primera pero en orden inverso.

```

1  Lista<int> Invertir(Lista <int> &L){
2      Lista<int>::Posicion p;
3      Lista<int> nueva;
4      for (p=L.begin();p!=L.end();++p)
5          nueva.Insertar(nueva.begin(),*p);
6
7      return nueva;
8  }

```

□

**Ejemplo 4.5.3**

Redefinir la clase Posicion de lista para que itere sobre los elementos pares de la lista. También se debe implementar el método begin y end de lista. Puede que la representación de Posición tenga que cambiar.

```

1  class Lista{
2  ...
3      class Posicion{
4          private:
5              Celda * punt;
6              Celda *cabecera; //nos hara falta para saber cuando terminamos

```

```

7      public:
8          Posicion():punt(0){}
9          int & operator *(){
10             return punt->ele;
11         }
12         Posicion & operator ++(){
13             punt= punt->sig;
14             while (punt!=cabecera && (punt->ele %2)==1)
15                 punt=punt->sig;
16             return *this;
17         }
18         Posicion & operator --(){
19             punt=punt->ant;
20             while (punt!=cabecera && punt->ele%2==1)
21                 punt=punt->ant;
22             return *this;
23         }
24         bool operator ==(const Posicion & P){
25             return punt==P.punt;
26         }
27         bool operator !=(const Posicion &P){
28             return punt!=P.punt;
29         }
30         friend class Lista;
31     };
32
33     Posicion begin()const{
34         Posicion p;
35         p.punt = cab->sig;
36         p.cabecera=cab;
37         if (p.punt->ele%2==1) ++p;
38         return p;
39     }
40     Posicion end()const{
41         Posicion p;
42         p.punt=cab;
43         p.cabecera=cab;
44         return p;
45     }
46
47 };

```

Como se puede observar la representación de Posición añade un campo mas: cabecera. Este campo en los métodos begin y end de lista se inicia a cab de lista ( dirección de la celda cabecera). Este campo,

cabecera, nos hace falta para poder implementar el operador `--` y operador `++`.

□

#### Ejemplo 4.5.4

Dada una lista de listas de enteros L:

$$L = \left\{ \begin{array}{cccc} < 1 & 1 & 0 & 1 > \\ < 1 & 0 & 1 & 1 > \\ < 0 & 1 & 1 & 1 > \\ < 1 & 1 & 1 & 0 > \end{array} \right\}$$

En la que cada lista dentro de L tiene el mismo numero de elementos. Construir una función para indicar si la suma por filas son iguales y además cada columna tambien suma lo mismo. En el ejemplo se puede observar que cada fila suma 3 y cada columna suma 3. Usad para implementar la función objetos de tipo Posición.

```

1  bool SumaCol_Filas(Lista<Lista<int> > &L){
2      vector< Lista<int>::Posicion> its=vector<Lista<int>::Posicion >(L.size());
3      vector<int> row_s=vector<int>(L.size(),0);
4      Lista< Lista<int> >::Posicion p;
5      //averiguamos cuantos elementos (columnas)tiene cada lista
6      int ncols=0;
7      p=L.begin();
8      for (Lista<int>::Posicion aux = (*p).begin();aux!=(*p).end();++aux,++ncols);
9
10     //incialiamos el vector de sumas por columnas
11     vector<int>cols_s=vector<int>(ncols,0);
12
13     //inicializamos los iteradores por fila
14     for (p=L.begin();p!=L.end();++p,++i){
15         its [i]=(*p).begin();
16     }
17     i=0;
18     while (its[0]!=(*(L.begin())).end()){
19
20         for (int j=0;j<L.size();j++){
21             cols_s[i]+=*(its[j]);
22             rows_s[j]+=*(its[j]);
23             ++(its[j]);//avanzamos el iterador
24         }
25         i++;
26     }
27
28
29

```

```

30     //Comprobamos que las sumas sean iguales
31     //la suma por columna coincidan
32     for (int j=1;j<cols_s.size();j++)
33         if (cols_s[j]!=cols_s[0]) return false;
34     //Comprobamos que las sumas por filas y columnas sean iguales
35     for (int j=0;j<rows_s.size();j++)
36         if (rows_s[j]!=cols_s[0]) return false;
37
38     return true;
39 }

```

□

## 4.6 Abstracción por iteración

A continuación vamos a formalizar el concepto que hemos visto tras el T.D.A. Posición del T.D.A Lista. Los *contenedores* son estructuras de datos que contienen almacenados una colección de elementos de un determinado tipo. Los contenedores que hemos visto son:

1. Vectores dinámicos
2. Pilas
3. Colas
4. Colas con prioridad
5. Listas

De entre estos contenedores en dos de ellos, Vectores dinámicos y Listas se pueden acceder a cualquier elemento en cualquier posición. Por ello para estos tipos de contenedores es interesante proponer una abstracción que permita recorrerlos de manera genérica. Con tal fin ya hemos visto para las listas el T.D.A Posición que no es más que una aproximación al concepto de *iterador*.

Los iteradores son un T.D.A. que actúa como un mecanismo para acceder a los elementos de un contenedor de una forma parecida a la forma de actuar de los punteros. Los pasos a seguir para trabajar con iteradores son:

1. Iniciar el iterador a la primera posición del contenedor (función begin()).
2. Acceder al elemento que apunta (\*it, donde it es de tipo iterador)
3. Avanzar el iterador al siguiente elemento del contenedor (++it)
4. Saber cuando hemos recorrido todos los elementos del contenedor (función end()).

Ahora nuestra clase Posición estará dentro de la clase y se llamará iterator. La estructura general sería:

```

1  class Lista {
2  private:
3      ...
4  public:
5      ...
6      class iterator {
7          ...
8      }

```

```

9
10     iterator begin() {
11         ...
12     }
13
14     iterator end() {
15         ...
16     }
17 }

```

Así, en el main podríamos trabajar de la siguiente manera:

```

1  Lista l;                //creamos nuestra lista
2  ...                    //ejecutamos mas instrucciones
3  Lista::iterator it;     //Creamos nuestro iterador
4  //Recorremos la lista
5  for (it=l.begin();it!=l.end();++it)
6      cout << *it;

```

## 4.7 Listas con celdas doblemente enlazadas y circulares con cabecera

Vamos a reescribir nuestra Lista vista en la sección 4.5.6 con el concepto de iterador.

### 4.7.1 Implementación

```

1  //Lista.h
2  template <class T>
3  struct Celda {
4      T d;
5      Celda<T> *sig, *ant;
6  };
7  template <class T>
8  class Lista {
9  private:
10     Celda<T> *primera; //cabecera
11
12 public:
13     Lista () {primera=0;};
14     Lista (const Lista<T> &l);
15     ~Lista ();
16     Lista & operator= (const Lista<T> &l);
17
18     class const_iterator; //debemos declara de forma adelantada esta clase para
19                           //avisar a iterator de su existencia
20     class iterator {

```

```

21     private:
22         Celda<T> *punt;
23
24     public:
25         iterator (): punt(0) {}
26         iterator & operator++ () {
27             punt=punt->sig;
28             return *this;
29         }
30         iterator & operator-- () {
31             punt=punt->ant;
32             return *this;
33         }
34         T & operator* () {return punt->d;}
35         bool operator== (const iterator &i) const {
36             return punt==i.punt;
37         }
38         bool operator!= (const iterator &i) const {
39             return punt!=i.punt;
40         }
41         friend class Lista;
42         friend class const_iterator;
43     };
44
45     class const_iterator {
46     private:
47         Celda<T> *punt;
48
49     public:
50         const_iterator(): punt(0) {}
51         //podemos construir const_iterator a partir de
52         //iteradores no constantes
53         const_iterator (const iterator &i) {
54             punt = i.punt;
55         }
56         bool operator== (const const_iterator &i) const {
57             return punt!=i.punt;
58         }
59         bool operator!= (const const_iterator &i) const {
60             return punt != i.punt;
61         }
62         const T & operator* () const {
63             return punt->d;

```

```

64         }
65         const_iterator & operator++ () {
66             punt = punt->sig;
67             return *this;
68         }
69         const_iterator & operator-- () {
70             punt = punt->ant;
71             return *this;
72         }
73         friend class Lista;
74     };
75
76     void Set (iterator it, const T& v) {
77         (*it) = v;
78     }
79     T Get (iterator it) const {
80         return *it;
81     }
82     void Insertar (iterator it, const T & e);
83     void Borrar (iterator it);
84
85     //begin y end para iterator
86     iterator begin() {
87         iterator it;
88         it.punt = primera->sig; //devolvemos la siguiente
89         return it; //a la cabecera
90     }
91     iterator end() {
92         iterator it;
93         it.punt = primera; //devolvemos la cabecera
94         return it;
95     }
96     //begin y end para const_iterator
97     const_iterator begin() const {
98         const_iterator it;
99         it.punt = primera->sig; //devolvemos la siguiente
100        return it; //a la cabecera
101    }
102    const_iterator end() const {
103        const_iterator it;
104        it.punt = primera; //devolvemos la cabecera
105        return it;
106    }

```



```

107 };
108 #include "Lista.cpp"

96 //Lista.cpp
97 void Lista<T>::Insertar (iterator it, const T & e) {
98     //it apunta a la celda donde queremos insertar la nuestra
99     Celda *aux = new Celda;           //Creamos una nueva celda
100     aux->d = e;                        //Le insertamos el valor
101     aux->sig = it.punt;                //La siguiente a la nuestra es
102                                     //la que ocupa la posicion que vamos a insertar
103     aux->ant = it.punt->ant;            //la anterior, la anterior de it
104     it.punt->ant->sig = aux;            //y la siguiente de la anterior, la nuestra
105     it.punt->ant = aux;                //y la siguiente a la nuestra, it
106                                     //es la anterior a it.
107 }
108
109 void Lista<T>::Borrar (iterator it) {
110     it.punt->ant->sig = it.punt->sig;    //La siguiente de la anterior es
111                                     //la siguiente de la que queremos borrar
112     it.punt->sig->ant = it.punt->ant;    //El anterior de la siguiente es
113                                     //la anterior de la que queremos borrar
114     delete it.punt;
115 }

```

**¿Por qué definimos dos iterators?** Porque debemos tener uno para tratar con listas constantes y otro para tratar con listas no constantes, ya que por ejemplo, en el siguiente código obtendríamos un error de compilación: se usa un `Lista<int>::iterator` en vez de `Lista<int>::const_iterator`. Con `Lista<int>::const_iterator` se preservan los valores de los elementos en la lista. En cambio usando `Lista<int>::iterator` se podría intentar, usando el operador `*`, cambiar algún dato de la lista. Tened en cuenta que el operador `*` devuelve el elemento de la lista por referencia.



```

1 #include "Lista.h"
2 void Imprimir (const Lista<int> &l) {
3     Lista<int>::iterator it;
4     for (it=l.begin(); it!=l.end(); ++it)
5         cout << *it;
6 }

```

#### 4.7.2 Ejemplos de funciones para nuestra lista

Con nuestra lista ya definida podríamos definir funciones template para hacer algunas cosas con ellas.

**Ejemplo 4.7.1**

Definir una función template para imprimir listas de cualquier tipo

```

1 //principal.cpp
2 #include "Lista.h"
3 #include <iostream>
4 using namespace std;
5
6 template <class T>
7 void Imprimir(const Lista<T> &l) {
8     /*Lista<T>::const_iterator it; nos daria error
9     de compilacion porque el compilador piensa que
10     es una definicion estatica dentro de lista. Para
11     evitar este error, definimos nuestro iterador con
12     typename*/
13
14     typename Lista<T>::const_iterator it;
15
16     for (it=l.begin();it!=l.end();++it)
17         cout << *it;
18 }
```

□

Cuando definimos una variable de un tipo, en este caso iterator, dentro de otro Lista<T>, que es template, hay que anteponer *typename*. En caso contrario se supone que se está accediendo a un miembro estático dentro de la clase Lista.

**Ejemplo 4.7.2**

Definir una función para imprimir los elementos de nuestra lista al revés

```

19 //Seguimos en principal.cpp
20 template <class T>
21 void Imprimir_invertido (const Lista<T> &l) {
22     typename Lista<T>::const_iterator it=l.end();
23     --it; //pasamos de la cabecera a la ultima celda
24     for (;it!=l.end();--it)
25         cout << *p;
26 }
```

□

**Ejemplo 4.7.3**

Definir una función que elimine los pares de una lista de enteros

```
27 //Principal.cpp
28 template <class T>
29 void EliminaPares (Lista<int> &l)
30 {
31     Lista<int>::iterator it;
32     //Aqui no hace falta typename porque ya tenemos la lista definida
33     it=l.begin();
34
35     while (it!=l.end()) {
36         if ((*it)%2==0)
37             it=l.Borrar(it); //La funcion borrar nos devuelve un iterador
38                             //Para no perder el iterador tras borrar el elemento
39
40         else
41             ++it;
42     }
43 }
```

□

#### Ejemplo 4.7.4

Definir una función que nos de información sobre cualquier contenedor

```
44 template <class InputIterator>
45 void ImprimirContenedor (const InputIterator &first, const InputIterator &last) {
46     InputIterator it;
47     for (it=first;it!=last;++it)
48         cout << *it;
49 }
```

□

Con las anteriores funciones podríamos tener un programa principal de la siguiente forma

```
50 //Principal.cpp
51
52 int main() {
53     Lista<int> l;
54
55     for (int i=0; i<10; i++)
56         l.Insertar(l.end(),i);
57
58     Imprimir(l);
59     Imprimir_invertido(l);
60     ImprimirContenedor(l.begin(),l.end());
}
```

```
61
62     vector<char> mivector;
63
64     for (char c='a';c!='z';c++)
65         mivector.push_back(c);
66
67     ImprimirContenedor(mivector.begin(),mivector.end());
68     //Usamos la misma funcion para lista y vector
69
70     return 0;
71 }
```