INTRODUCCIÓN

En este capítulo se comienza el estudio de las estructuras de datos dinámicas. Al contrario que las estructuras de datos estáticas (*arrays* —listas, vectores y tablas— y *estructuras*) en las que su tamaño en memoria se establece durante la compilación y permanece inalterable durante la ejecución del programa, las estructuras de datos dinámicas crecen y se contraen a medida que se ejecuta el programa.

La estructura de datos que se estudiará en este capítulo es la **lista enlazada** (ligada o encadenada, "*linked list*") que es una colección de elementos (denominados nodos) dispuestos uno a continuación de otro, cada uno de ellos conectado al siguiente elemento por un "enlace" o "referencia". En el capítulo se desarrollan algoritmos para insertar, buscar y borrar elementos en las listas enlazadas. De igual modo, se muestra el tipo abstracto de datos (*TAD*) que representa a las listas enlazadas.

10.1. FUNDAMENTOS TEÓRICOS DE LISTAS ENLAZADAS

Las estructuras lineales de elementos homogéneos (listas, tablas, vectores) implementadas con *arrays* necesitan fijar por adelantado el espacio a ocupar en memoria, de modo que cuando se desea añadir un nuevo elemento, que rebase el tamaño prefijado, no será posible sin que se produzca un error en tiempo de ejecución. Esto hace ineficiente el uso de los arrays en algunas aplicaciones.

Gracias a la asignación dinámica de memoria, se pueden implementar listas de modo que la memoria física utilizada se corresponda exactamente con el número de elementos de la tabla. Para ello se recurre a los punteros (*apuntadores*) y *variables apuntadas* que se crean en tiempo de ejecución.

Una **lista enlazada** es una colección o secuencia de elementos dispuestos uno detrás de otro, en la que cada elemento se conecta al siguiente elemento por un "enlace". La idea básica consiste en construir una lista cuyos elementos, llamados **nodos**, se componen de dos partes (*campos)*: la primera parte contiene la información y es, por consiguiente, un valor de un tipo genérico (denominado *Dato, TipoElemento*, *Info*, etc.), y la segunda parte es un *enlace* que apunta al siguiente nodo de la lista.

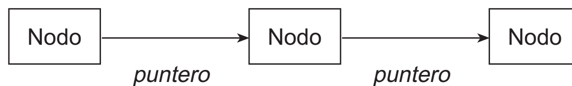
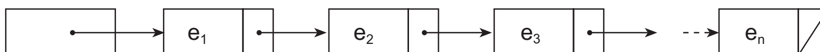


Figura 10.1. Lista enlazada (representación simple).

La representación gráfica más extendida es aquella que utiliza una caja con dos secciones en su interior. En la primera sección se encuentra el elemento o valor del dato y en la segunda sección el enlace, representado mediante una flecha que sale de la caja y apunta al siguiente nodo.



e_1, e_2, ...e_n son valores del tipo `TipoElemento`

Figura 10.2. Lista enlazada (representación gráfica típica).

www.FreeLibros.me

Definición

Una **lista enlazada** consta de un número de nodos con dos componentes (*campos*), un enlace al siguiente nodo de la lista y un v alor, que puede ser de cualquier tipo

Los enlaces se representan por flechas para facilitar la comprensión de la conexión entre dos nodos; ello indica que el enlace tiene la dirección en memoria del siguiente nodo. Los enlaces también sitúan los nodos en una secuencia. La Figura 10.2 muestra una lista cuyos nodos forman una secuencia desde el primer elemento (e_1) al último elemento (e_n). El último nodo ha de representarse de forma diferente, para significar que este nodo no se enlaza a ningún otro. La Figura 10.3 muestra diferentes representaciones gráficas que se utilizan para dibujar el campo enlace del último nodo.

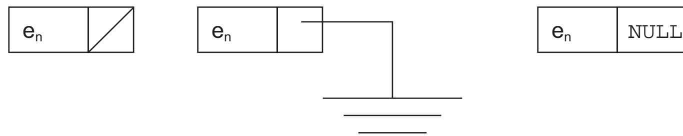


Figura 10.3. Diferentes representaciones gráficas del nodo último.

10.1.1. Clasificación de las listas enlazadas

Las listas se pueden dividir en cuatro categorías:

- *Listas simplemente enlazadas*. Cada nodo (elemento) contiene un único enlace que conecta ese nodo al nodo siguiente o nodo sucesor. La lista es eficiente en recorridos directos (“adelante”).
- *Listas doblemente enlazadas*. Cada nodo contiene dos enlaces, uno a su nodo predecesor y el otro a su nodo sucesor. La lista es eficiente tanto en recorrido directo (“adelante”) como en recorrido inverso (“atrás”).
- *Lista circular simplemente enlazada*. Una lista simplemente enlazada en la que el último elemento (cola) se enlaza al primer elemento (cabeza) de tal modo que la lista puede ser recorrida de modo circular (“en anillo”).
- *Lista circular doblemente enlazada*. Una lista doblemente enlazada en la que el último elemento se enlaza al primer elemento y viceversa. Esta lista se puede recorrer de modo circular (en anillo) tanto en dirección directa (“adelante”) como inversa (“atrás”).

La implementación de cada uno de los cuatro tipos de estructuras de listas se puede desarrollar utilizando punteros.

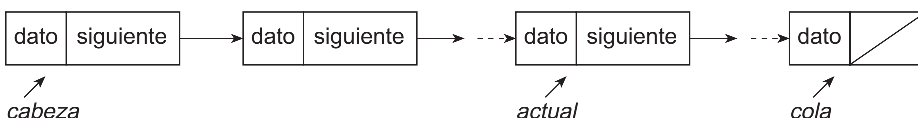


Figura 10.4. Representación gráfica de una lista enlazada.

www.FreeLibros.me

El primer nodo, **frente**, de una lista es el nodo apuntado por **cabeza**. La lista encadena nodos juntos desde el frente al final (**cola**) de la lista. El final se identifica como el nodo cuyo campo enlace tiene el valor `NULL`. La lista se recorre desde el primero al último nodo; en cualquier punto del recorrido la posición *actual* se referencia por el puntero `actual`. Una lista vacía, es decir, que no contiene nodos se representa con el puntero `cabeza` a nulo.

10.2. TIPO ABSTRACTO DE DATOS *Lista*

Una lista almacena información del mismo tipo, con la característica de que puede contener un número indeterminado de elementos, y que estos elementos mantienen un orden explícito. Este ordenamiento explícito se manifiesta en que, en sí mismo, cada elemento contiene la dirección del siguiente elemento. Cada elemento es un nodo de la lista.

Una lista es una estructura de datos dinámica. El número de nodos puede variar rápidamente en un proceso. Aumentando los nodos por inserciones, o bien, disminuyendo por eliminación de nodos.

Las inserciones se pueden realizar por cualquier punto de la lista. Por la cabeza (inicio), por el final (cola), a partir o antes de un nodo determinado de la lista. Las eliminaciones también se pueden realizar en cualquier punto de la lista; además se eliminan nodos dependiendo del campo de información o dato que se desea suprimir de la lista.

10.2.1. Especificación formal del *TAD Lista*

Matemáticamente, una lista es una secuencia de cero o más elementos de un determinado tipo.

```
(a1, a2, a3, ... , an)     donde n >= 0, si n = 0 la lista es vacía.
```

Los elementos de la lista tienen la propiedad de que sus elementos están ordenados de forma lineal, según las posiciones que ocupan en la misma. Se dice que a_i precede a a_{i+1} para `i = 1 ..., n-1`; y que a_i sucede a a_{i-1} para `i = 2 ... n`.

Para formalizar el *Tipo Abstracto de Dato Lista* a partir de la noción matemática, se define un conjunto de operaciones básicas con objetos de tipo `Lista`. Las operaciones:

```
∀ L ∈ Lista,      ∀ x ∈ Dato,   ∀ p ∈ puntero
```

`Listavacia(L)`	Inicializa la lista `L` como lista vacía.
`Esvacia(L)`	Determina si la lista `L` está vacía.
`Insertar(L,x,p)`	Inserta en la lista `L` un nodo con el campo dato `x`, delante del nodo de dirección `p`.
`Localizar(L,x)`	Devuelve la posición/dirección donde está el campo de información `x`.
`Suprimir(L,x)`	Elimina de la lista el nodo que contiene el dato `x`.
`Anterior(L,p)`	Devuelve la posición/dirección del nodo anterior a `p`.
`Primero(L)`	Retorna la posición/dirección del primer nodo de la lista `L`.
`Anula(L)`	Vacía la lista `L`.

www.FreeLibros.me

Estas operaciones son las básicas para manejar listas. En realidad, la decisión de qué operaciones son las básicas depende de las características de la aplicación que se va a realizar con los datos de la lista. También dependerá del tipo de representación elegido para las listas. Así, para añadir nuevos nodos a una lista se implementan, además de `insertar()`, versiones de ésta como:

`inserPrimero(L,x)` Inserta un nodo con el dato `x` como primer nodo de la lista `L`.
`inserFinal(L,x)` Inserta un nodo con el dato `x` como último nodo de la lista `L`.

Otra operación típica de toda estructura enlazada de datos es *recorrer*. Consiste en visitar cada uno de los datos o nodos de que consta. En las listas enlazadas, normalmente se realiza desde el nodo *cabeza* al último nodo o *cola* de la lista.

10.3. OPERACIONES DE LISTAS ENLAZADAS

La implementación del *TAD* `Lista` requiere, en primer lugar, declarar la clase `Nodo` en la cual se *encierra* el dato (entero, real, doble, carácter, o referencias a objetos) y el enlace. Además, la clase `Lista` con las operaciones (*creación, inserción, ...*) y el atributo *cabeza* de la lista.

10.3.1. Clase *Nodo*

Una lista enlazada se compone de una serie de nodos enlazados mediante punteros. La clase `Nodo` declara las dos partes en que se divide: `dato` y `enlace`. Además, el constructor y la interfaz básica; por ejemplo, para el caso de una lista enlazada de números enteros:

```
typedef int Dato;
        // archivo de cabecera Nodo.h
#ifndef _NODO_H
#define _NODO_H
class Nodo
{
protected:
    Dato dato;
    Nodo* enlace;
public:
  Nodo(Dato t)
  {
    dato = t;
    enlace = 0;                // 0 es el puntero NULL en C++
  }
  Nodo(Dato p, Nodo* n)        // crea el nodo y lo enlaza a n
  {
    dato = p;
    enlace = n;
  }

  Dato datoNodo() const
```

www.FreeLibros.me

```
    {
      return dato;
    }

    Nodo* enlaceNodo() const
    {
      return enlace;
    }

    void ponerEnlace(Nodo* sgte)
    {
      enlace = sgte;
    }
};
#endif
```

Dado que los datos que se puede incluir en una lista pueden ser de cualquier tipo (entero, real, caracter o cualquier objeto), puede declararse un nodo genérico y, en consecuencia, una lista genérica con las plantillas de C++:

```
template <class T> class NodoGenerico
{
protected:
    T dato;
    NodoGenerico <T>* enlace;

public:
    NodoGenerico (T t)
    {
      dato = t;
      enlace = 0;
    }
    NodoGenerico (T p, NodoGenerico<T>* n)
    {
      dato = p;
      enlace = n;
    }

    T datoNodo() const
    {
       return dato;
    }

    NodoGenerico<T>* enlaceNodo() const
    {
       return enlace;
    }

    void ponerEnlace(NodoGenerico<T>* sgte)
    {
      enlace = sgte;
    }
};
```

www.FreeLibros.me

EJEMPLO 10.1. Se declara la clase `Punto` para representar un punto en el plano , con su coordenada x e y. También, se declara la clase `Nodo` con el campo dato de tipo `Punto`.

```
// archivo de cabecera punto.h
class Punto
{
  double x, y;

  public:
    Punto(double x = 0.0, double y = 0.0)
    {
      this → x = x;
      this → y = y;
    }
};
```

La declaración de la clase `Nodo` consiste, sencillamente, en asociar el nuevo tipo de dato, el resto no cambia.

```
typedef Punto Dato;
#include "Nodo.h"
```

10.3.2. Acceso a la lista: *cabecera* y *cola*

Cuando se construye y utiliza una lista enlazada en una aplicación, el acceso a la lista se hace mediante uno, o más, *punteros* a los nodos. Normalmente, se accede a partir del primer nodo de la lista, llamado **cabeza** o **cabecera** de la lista. En ocasiones, se mantiene también un apuntador al último nodo de la lista enlazada, llamado **cola** de la lista.

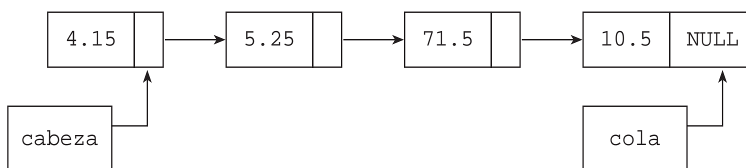


Figura 10.5. Declaraciones de tipos en lista enlazada.

Los apuntadores, `cabeza` y `cola`, se declararan como variables puntero a `Nodo`:

```
Nodo* cabeza;
Nodo* cola;
```

La Figura 10.5 muestra una lista a la que se accede con el puntero `cabeza`; cada nodo está enlazado con el siguiente nodo. El último nodo, cola o final de la lista, no se enlaza con otro nodo, entonces su campo de enlace contiene *nulo* (*0* o `NULL` indistintamente). Normalmente `NULL` se utiliza en dos situaciones:

www.FreeLibros.me

- Campo *enlace* del último nodo (*final o cola*) de una lista enlazada.
- Como valor `cabeza`, para una lista enlazada que no tiene nodos, es decir una **lista vacía** `(cabeza = NULL)`.

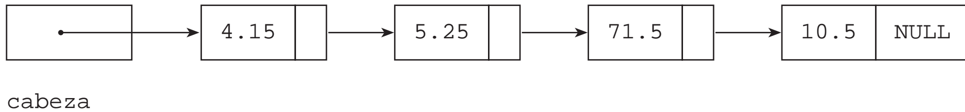


Figura 10.6. Acceso a una lista con puntero cabeza.

Nota de programación

Las variables de acceso a una lista, `cabeza` y `cola`, se inicializan a `NULL` cuando comienza la construcción de la lista.

Error de programación

Uno de los errores típicos en el tr atamiento de punteros consiste en escr ibir la expresión `p → miembro` cuando el valor del puntero `p` es `NULL`.

10.3.3. Clase `Lista`: construcción de una lista

La creación de una lista enlazada implica la definición de, al menos, la clases `Nodo` y `Lista`. La clase `Lista` contiene el puntero de acceso a la lista enlazada, de nombre primero, que apunta al nodo cabeza; también se puede declarar un puntero al nodo cola, como no se necesita para implementar las operaciones no se ha declarado.

Las funciones de la clase `Lista` implementan las operaciones de una lista enlazada: *inserción*, *búsqueda* .... El constructor inicializa `primero` a `NULL`, (*lista vacía*). Además, `crearLista()` construye iterativamente el primer elemento (`primero`) y los elementos sucesivos de una lista enlazada.

El Ejemplo 10.2 declara una lista para un tipo particular de dato: `int`. Lo más interesante del ejemplo es la codificación, paso a paso, de la función `crearLista()`.

EJEMPLO 10.2. Crear una lista enlazada de elementos que almacenen datos de tipo entero.

La declaración de la clase `Nodo` se encuentra en el archivo de cabecera `Nodo.h`, (*consultar* Apartado 10.3.1).

```
typedef int Dato;
#include "Nodo.h"
```

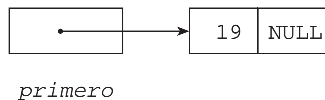
www.FreeLibros.me

```
class Lista
{
protected:
    Nodo* primero;
  public:
    Lista()
     {
       primero = NULL;
    }
       void crearLista();
    //...
```

La referencia `primero` (también se puede llamar `cabeza`) se ha inicializado en el constructor a un valor *nulo*, es decir, a *lista vacía*.

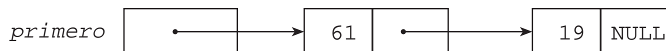
A continuación, y antes de escribir su código, se muestra el comportamiento de la función `crearLista()`. En primer lugar, se crea un nodo con un valor y su dirección se asigna a `primero`:

```
primero = new Nodo(19);
```



Ahora se desea añadir un nuevo elemento con el valor `61`, y situarlo en el primer lugar de la lista. Se utiliza el constructor de `Nodo` que enlaza con otro nodo ya creado:

```
primero = new Nodo(61,primero);
```



Por último, para obtener una lista compuesta de `4, 61, 19` se habría de ejecutar:

```
primero = new Nodo(4,primero);
```

A continuación, se escribe `CrearLista()` que codifica las acciones descritas anteriormente. Los valores se leen del teclado, termina con el valor clave `-1`.

```
void Lista::crearLista()
{
  int x;
  primero = 0;
  cout << "Termina con -1" << endl;
  do {
    cin >> x;
    if (x != -1)
```

www.FreeLibros.me

```
        {
          primero = new Nodo(x,primero);
        }
      }while (x != -1);
  }
```

10.4. INSERCIÓN EN UNA LISTA

El nuevo elemento que se desea incorporar a una lista se puede insertar de distintas formas, según la posición o punto de inserción. Éste puede ser:

- En la cabeza (elemento primero) de la lista.
- En el final o cola de la lista (elemento último).
- Antes de un elemento especificado, o bien.
- Después de un elemento especificado.

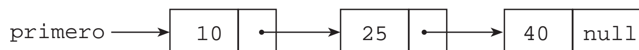
10.4.1. Insertar en la cabeza de la lista

La posición más facil y, a la vez, más eficiente donde insertar un nuevo elemento en una lista es por la *cabeza*. El proceso de inserción se resume en este algoritmo:

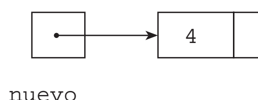
1. Crear un nodo e inicializar el campo dato al nuevo elemento. La dirección del nodo creado se asigna a `nuevo`.
2. Hacer que el campo `enlace` del nodo creado apunte a la *cabeza* (`primero`) de la lista.
3. Hacer que `primero` apunte al nodo que se ha creado.

El Ejemplo 10.3 inserta un elemento por la *cabeza* de una lista siguiendo los pasos del algoritmo y se escribe el código.

EJEMPLO 10.3. Una lista enlazada contiene tres elementos , 10, 25 y 40. Insertar un nuevo elemento, 4, en cabeza de la lista.

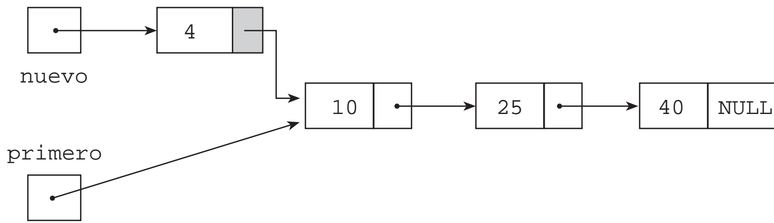


Paso 1

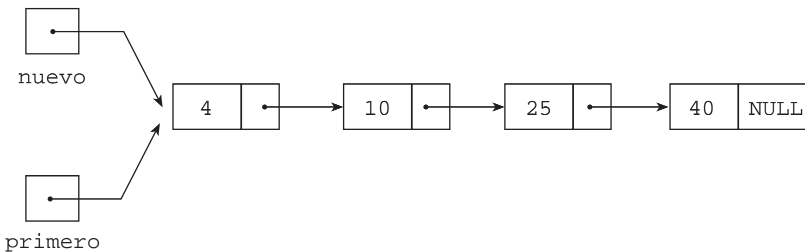


www.FreeLibros.me

Paso 2



Paso 3



En este momento, la función termina su ejecución, la variable local `nuevo` desaparece y sólo permanece el puntero `primero` al inicio de la lista.

El código fuente de `insertarCabezaLista:`

```
void Lista::insertarCabezaLista(Dato entrada)
{
  Nodo* nuevo ;
  nuevo = new Nodo(entrada);
  nuevo -> ponerEnlace(primero);       // enlaza nuevo con primero
  primero = nuevo;     // mueve primero y apunta al nuevo nodo
}
```

Caso particular

`insertarCabezaLista` también actúa correctamente si se trata de añadir un primer nodo a una lista vacía. En este caso, `primero` apunta a `NULL` y termina apuntando al nuevo nodo de la lista enlazada.

EJERCICIO 10.1. Programa para crear una lista de números aleatorios. Inserta los nuevos nodos por la cabeza de la lista. Un vez creada la lista, se ha de recorrer para mostrar los número pares.

Se crea una lista enlazada de números enteros, para ello se utilizan las clases `Nodo` y `Lista` según las declaraciones de los Apartado 10.3 y 10.4. En esta última se añade la función `visualizar()` que recorre la lista escribiendo el campo dato de cada nodo. Desde `main()` se

www.FreeLibros.me

crea un objeto `Lista`, se llama iterativamente a la función miembro `insertarCabezaLista`, y, por último, se llama a `visualizar()` para mostrar los elementos.

```
// archivo de cabecera Lista.h e implementación de visualizar()

#include <iostream >
using namespace std;
typedef int Dato;
#include "Nodo.h"

class Lista
{
protected:
    Nodo* primero;
public:
    Lista();
    void crearLista();
    void insertarCabezaLista(Dato entrada);
    void visualizar();
};

void Lista::visualizar()
{
  Nodo* n;
  int k = 0;
  n = primero;
  while (n != NULL)
  {
    char c;
    k++; c = (k%15 != 0 ? ' ' : '\n');
    cout << n -> datoNodo() << c;
    n = n -> enlaceNodo();
  }
}

// archivo con la función main
#include <iostream >
using namespace std;
typedef int Dato;
#include "Nodo.h"
#include "Lista.h"

int main()
{
  Dato d;
  Lista lista;     // crea lista vacía
  cout << "Elementos de la lista, termina con -1 " << endl;
  do {
    cin >> d;
    lista.insertarCabezaLista(d);
  } while (d != -1);
  // recorre la lista para escribir sus elementos
  cout << "\t Elementos de la lista generados al azar" << endl;
  lista.visualizar();
  return 0;
}
```

www.FreeLibros.me

10.4.2. Inserción al final de la lista

La inserción al final de la lista es menos eficiente debido a que, normalmente, no se tiene un puntero al último nodo y entonces se ha de seguir la traza desde la cabeza de la lista hasta el último nodo y, a continuación, realizar la inserción. Una vez que la variable `ultimo` apunta al final de la lista, el enlace con el nuevo nodo es sencillo:

```
ultimo -> ponerEnlace(new Nodo(entrada));
```

El campo `enlace` del último nodo queda apuntando al nodo creado y así se enlaza, como nodo final, a la lista.

A continuación, se codifica la función *public* `insertarUltimo()` junto a la función que recorre la lista y devuelve el puntero al último nodo.

```
void Lista::insertarUltimo(Dato entrada)
{
  Nodo* ultimo = this -> ultimo();
  ultimo -> ponerEnlace(new Nodo(entrada));
}

Nodo* Lista::ultimo()
{
  Nodo* p = primero;
  if (p == NULL ) throw "Error, lista vacía";
  while (p -> enlaceNodo()!= NULL) p = p -> enlaceNodo();
  return p;
}
```

10.4.3. Insertar entre dos nodos de la lista

La inserción de nodo no siempre se realiza al principio o al final de la lista, puede hacerse entre dos nodos cualesquiera. Por ejemplo, en la lista de la Figura 10.7 se quiere insertar el elemento `75` entre los nodos con los datos `25` y `40`.

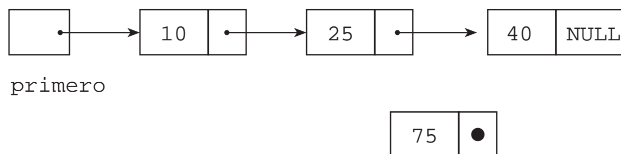


Figura 10.7. Inserción entre dos nodos.

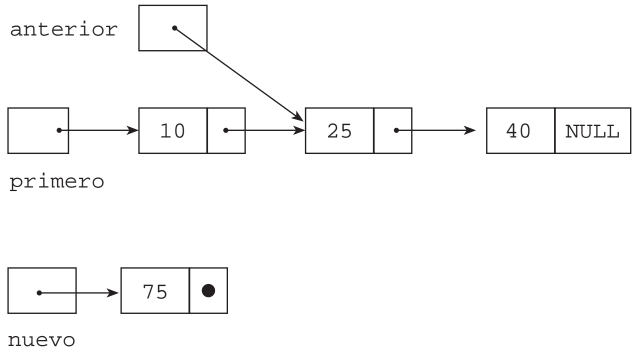
El algoritmo para la operación de insertar entre dos nodos (`n1, n2`) requiere las siguientes etapas:

1. Crear un nodo con el elemento y el campo `enlace` a `NULL`.
2. Poner campo `enlace` del nuevo nodo apuntando a `n2`, ya que el nodo creado se ubicará justo antes de `n2`.
3. Si el puntero `anterior` tiene la dirección del nodo `n1`, entonces poner su atributo `enlace` apuntando al nodo creado.

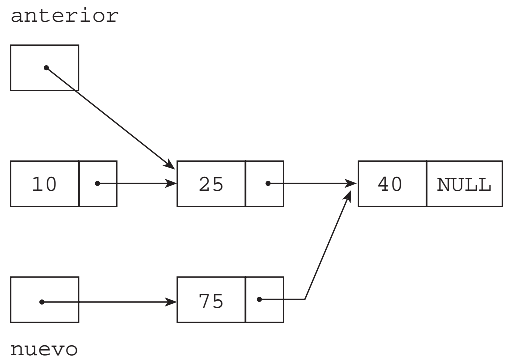
www.FreeLibros.me

A continuación se muestra gráficamente las etapas del algoritmo relativas a la inserción de `75` entre `25` (`n1`) y `40` (`n2`).

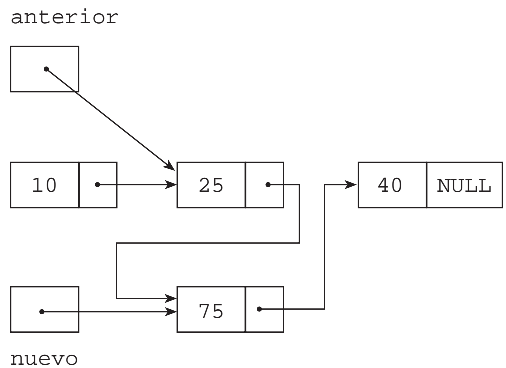
Etapas 1



Etapa 2



Etapa 3



www.FreeLibros.me

La operación es una función miembro de la clase *Lista*:

```
void Lista::insertarLista(Nodo* anterior, Dato entrada)
{
  Nodo* nuevo;

  nuevo = new Nodo(entrada);
  nuevo -> ponerEnlace(anterior -> enlaceNodo());
  anterior -> ponerEnlace(nuevo);
}
```

Antes de llamar a `insertarLista()`, es necesario buscar la dirección del nodo `n1`, esto es, del nodo a partir del cual se enlazará el nodo que se va a crear.

Otra versión de la función tiene como argumentos el dato a partir del cual se realiza el enlace, y el dato del nuevo nodo: `insertarLista(Dato testigo, Dato entrada)`. El algoritmo de este versión, primero busca el nodo con el dato *testigo* a partir del cual se inserta, y, a continuación, realiza los mismos enlaces que en la anterior función.

10. 5. BÚSQUEDA EN LISTAS ENLAZADAS

La operación *búsqueda* de un elemento en una lista enlazada recorre la lista hasta encontrar el nodo con el elemento. El algoritmo que se utiliza, una vez encontrado el nodo, devuelve el puntero al nodo (en caso negativo, devuelva `NULL`). Otro planteamiento consiste en devolver `true` si encuentra el nodo y `false` si no está en la lista

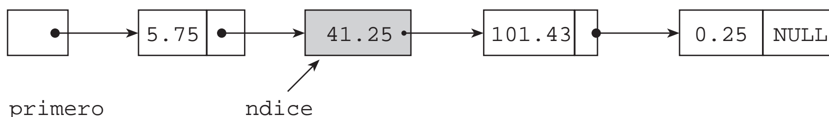


Figura 10.8. Búsqueda en una lista.

La función `buscarLista` de la clase `Lista` utiliza el puntero `indice` para recorrer la lista, nodo a nodo. Primero, se inicializa `indice` al nodo *cabeza* (`primero`), a continuación se compara el dato del nodo apuntado por `indice` con el elemento buscado, si coincide la búsqueda termina, en caso contrario `indice` avanza al siguiente nodo. La búsqueda termina cuando se encuentra el nodo, o bien cuando se ha terminado de recorrer la lista y entonces `indice` toma el valor `NULL`.

```
Nodo* Lista::buscarLista(Dato destino)
{
  Nodo* indice;

  for (indice = primero; indice!= NULL; indice = indice->enlaceNodo())
    if (destino == indice -> datoNodo())
      return indice;
  return NULL;
}
```

www.FreeLibros.me

EJEMPLO 10.4. Se escribe una función alternativa a la búsqueda del nodo que contiene un campo dato. Ahora también se devuelve un puntero a nodo, pero con el criterio de que ocupa una posición, pasada como argumento, en una lista enlazada.

La función es un miembro *público* de la clase `Lista`, por consiguiente, tiene acceso a sus miembros y dado que se busca por posición en la lista, se considera posición *1* la del nodo cabeza (`primero`); posición *2* al siguiente nodo, y así sucesivamente.

El algoritmo de búsqueda comienza inicializando `indice` al nodo cabeza de la lista. En cada iteración del bucle se mueve `indice` un nodo hacia adelante. El bucle termina cuando se alcanza la posición deseada, o bien si `indice` apunta a `NULL` como consecuencia de que la posición solicitada es mayor que el número de nodos de la lista.

```
Nodo* Lista::buscarPosicion(int posicion)
{
  Nodo* indice;
  int i;

  if (0 >= posicion)             // posición ha de ser mayor que 0
    return NULL;
  indice = primero;
  for (i = 1;(i < posicion) && (indice != NULL); i++)
    indice = indice -> enlaceNodo();
  return indice;
}
```

10.6. BORRADO DE UN NODO

Eliminar un nodo de una lista enlazada supone enlazar el nodo anterior con el nodo siguiente al que se desea eliminar y liberar la memoria que ocupa. El algoritmo se enfoca para eliminar un nodo que contiene un dato, sigue estos pasos:

1. Búsqueda del nodo que contiene el dato. Se ha de obtener la dirección del nodo a eliminar y la dirección del anterior.
2. El `enlace` del nodo anterior que apunte al nodo siguiente al que se elimina.
3. Si el nodo a eliminar es el *cabeza* de la lista (`primero`), se modifica `primero` para que tenga la dirección del siguiente nodo.
4. Por último, la memoria ocupada por el nodo se libera.

Naturalmente, `eliminar()` es una función miembro y *pública* de la clase `Lista`, recibe el dato del nodo que se quiere borrar. Desarrolla su propio bucle de búsqueda con el fin de disponer de la dirección del nodo anterior.

```
void Lista::eliminar(Dato entrada)
{
  Nodo *actual, *anterior;
  bool encontrado;

  actual = primero;
  anterior = NULL;
```

www.FreeLibros.me

```
    encontrado = false;
                              // búsqueda del nodo y del anterior
    while ((actual != NULL) && !encontrado)
    {
      encontrado = (actual -> datoNodo() == entrada);
      if (!encontrado)
      {
        anterior = actual;
        actual = actual -> enlaceNodo();
      }
    }
                              // enlace del nodo anterior con el siguiente
    if (actual != NULL)
    {
                           // distingue entre cabecera y resto de la lista
      if (actual == primero)
      {
        primero = actual -> enlaceNodo();
      }
      else
      {
        anterior -> ponerEnlace(actual -> enlaceNodo());
      }
      delete actual;
    }
  }
```

10.7. LISTA ORDENADA

Los elementos de una lista tienen la propiedad de estar ordenados de forma lineal, según las posiciones que ocupan en la misma. Se dice que n_i precede a n_{i+1} para `i = 1 ... n-1`; y que n_i sucede a n_{i-1} para `i = 2 ... n`. Ahora bien, tambien es posible mantener una lista enlazada ordenada según el dato asociado a cada nodo. La Figura 10.9 muestra una lista enlazada de números reales, ordenada de forma creciente.

La forma de insertar un elemento en una lista ordenada siempre realiza la operación de tal forma que la lista resultante mantiene la propiedad de ordenación. Para lo cual, en primer lugar, determina la posición de inserción y, a continuación, ajusta los enlaces.

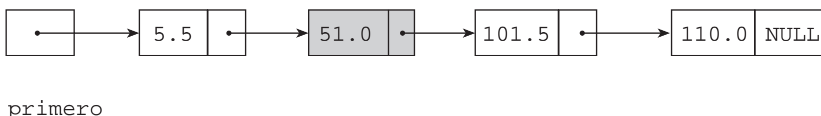


Figura 10.9. Lista ordenada.

Por ejemplo, para insertar el dato `104` en la lista de la Figura 10.9 es necesario recorrer la lista hasta el nodo con `110.0`, que es el nodo inmediatamente mayor. El puntero `indice` se queda con la dirección del nodo anterior, a partir del cual se realizan los enlaces con el nuevo nodo.

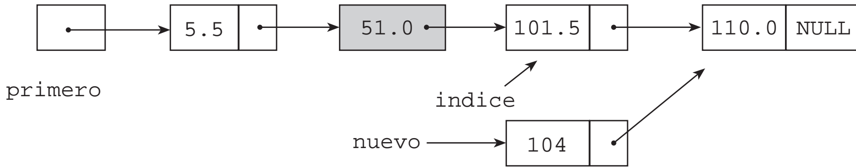


Figura 10.10. Inserción en una lista ordenada.

10.7.1. Clase `ListaOrdenada`

Una lista enlazada ordenada *es una* lista enlazada a la que se añade la propiedad ordenación de sus datos. Ésa es la razón que aconseja declarar la clase `ListaEnlazada` *derivada* de la clase `Lista`, por consiguiente, heredará las propiedades de `Lista`. Las funciones `eliminar()` y `buscarLista()` se deben redefinir para que los bucles de búsqueda aprovechen el hecho de que los datos están ordenados.

La función `insertaOrden()` crea la lista ordenada; el punto de partida es una lista vacía, a la que se añaden nuevos elementos, de tal forma que en todo momento los elementos están ordenados en orden creciente. La inserción del primer nodo de la lista consiste, sencillamente, en crear el nodo y asignar su dirección a la cabeza de la lista. El segundo elemento se ha de insertar antes o después del primero, dependiendo de que sea menor o mayor. El tipo de los datos de una lista ordenada han de ser ordinal, para que se pueda aplicar los operadores `==`, `<`, `>`. A continuación, se escribe el código que implementa la función.

```
void ListaOrdenada::insertaOrden(Dato entrada)
{
  Nodo* nuevo ;
  nuevo = new Nodo(entrada);
  if (primero == NULL)        // lista vacía
    primero = nuevo;
  else if (entrada < primero -> datoNodo())
  {
      nuevo -> ponerEnlace(primero);
      primero = nuevo;
  }
  else           // búsqueda del nodo anterior al de inserción
  {
    Nodo *anterior, *p;
    anterior = p = primero;

    while ((p->enlaceNodo()!= NULL) && (entrada > p->datoNodo()))
    {
        anterior = p;
        p = p -> enlaceNodo();
    }

    if (entrada > p->datoNodo()) // se inserta después del último
    anterior = p;
    // se procede al enlace del nuevo nodo
    nuevo -> ponerEnlace(anterior-> enlaceNodo());
```

www.FreeLibros.me

```
      anterior -> ponerEnlace(nuevo);
    }
  }
```

10.8. LISTA DOBLEMENTE ENLAZADA

Hasta ahora el recorrido de una lista se ha realizado en sentido directo (*adelante*). Existen numerosas aplicaciones en las que es conveniente poder acceder a los nodos de una lista en cualquier orden, tanto hacia adelante como hacia atrás. Desde un nodo de una **lista doblemente enlazada** se puede avanzar al siguiente, o bien retroceder al nodo anterior. Cada nodo de una lista doble tiene tres campos, el dato y dos punteros, uno apunta al siguiente nodo de la lista y el otro al nodo anterior. La Figura 10.11 muestra una lista doblemente enlazada y un nodo de dicha lista.

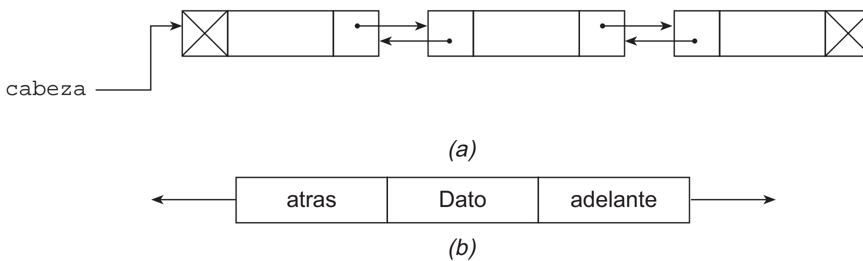


Figura 10.11. Lista doblemente enlazada. *(a)* Lista con tres nodos; *(b)* nodo.

Las operaciones de una *Lista Doble* son similares a las de una *Lista: insertar, eliminar, buscar, recorrer...* La operación de insertar un nuevo nodo en la lista debe realizar ajustes de los dos punteros. La Figura 10.12 muestra los movimientos de punteros para insertar un nodo, como se observa intervienen cuatro enlaces.

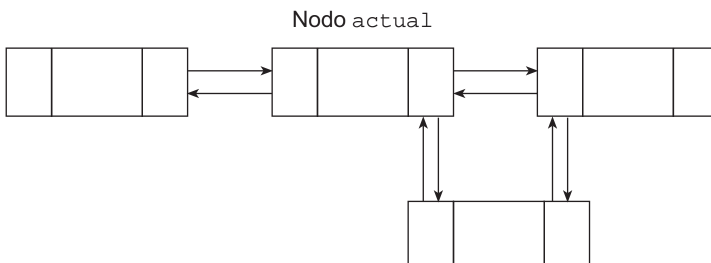


Figura 10.12. Inserción de un nodo en una lista doblemente enlazada.

La operación de eliminar un nodo de la lista doble necesita enlazar, mutuamente, el nodo anterior y el nodo siguiente del que se borra, como se observa en la Figura 10.13.

www.FreeLibros.me

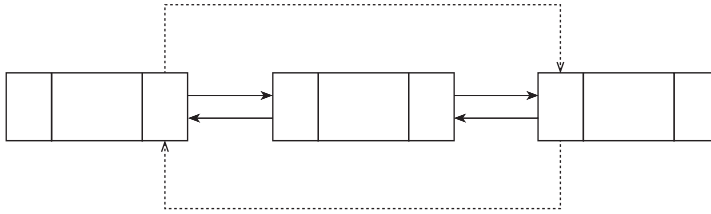


Figura 10.13. Eliminación de un nodo en una lista dob lemente enlazada.

10.8.1. Nodo de una lista doblemente enlazada

La clase `NodoDoble` agrupa los componentes del nodo de una lista doble y las operaciones de la *interfaz*.

```
// archivo de cabecera NodoDoble.h
class NodoDoble
{
protected:
    Dato dato;
    NodoDoble* adelante;
    NodoDoble* atras;

public:
    NodoDoble(Dato t)
    {
      dato = t;
      adelante = atras = NULL;
    }

    Dato datoNodo() const { return dato; }

    NodoDoble* adelanteNodo() const { return adelante; }
    NodoDoble* atrasNodo() const { return atras; }

    void ponerAdelante(NodoDoble* a) { adelante = a; }
    void ponerAtras(NodoDoble* a) { atras = a; }
};
```

10.8.2. Insertar un nodo en una lista doblemente enlazada

La clase `ListaDoble` encapsula las operaciones básicas de las listas doblemente enlazadas. La clase dispone del puntero variable `cabeza` para acceder a la lista, apunta al primer nodo. El constructor de la clase inicializa la lista vacía.

Se puede añadir nodos a la lista de distintas formas, según la posición donde se inserte. La posición de inserción puede ser:

- En *cabeza* de la lista.
- Al *final* de la lista.

www.FreeLibros.me

- Antes de un elemento especificado.
- Después de un elemento especificado.

Insertar por la cabeza

El proceso sigue estos pasos:

1. Crear un nodo con el nuevo elemento.
2. Hacer que el campo `adelante` del nuevo nodo apunte a la *cabeza* (primer nodo) de la lista original, y que el campo `atras` del nodo *cabeza* apunte al nuevo nodo.
3. Hacer que `cabeza` apunte al nodo creado.

A continuación, se escribe la función miembro de la clase `ListaDoble`, que implementa la operación.

```
void ListaDoble::insertarCabezaLista(Dato entrada)
{
  NodoDoble* nuevo;

  nuevo = new NodoDoble (entrada);
  nuevo -> ponerAdelante(cabeza);

  if (cabeza != NULL )
    cabeza -> ponerAtras(nuevo);
  cabeza = nuevo;
}
```

Insertar después de un nodo

El algoritmo de la operación que inserta un nodo después de otro, `n`, requiere las siguientes etapas:

1. Crear un nodo, `nuevo`, con el elemento.
2. Poner el enlace `adelante` del nodo creado apuntando al nodo siguiente de `n`. El enlace `atras` del nodo siguiente a `n` (si `n` no es el último nodo) tiene que apuntar a `nuevo`.
3. Hacer que el enlace `adelante` del nodo `n` apunte al nuevo nodo. A su vez, el enlace `atras` del nuevo nodo debe de apuntar a `n`.

La función `insertaDespues()` implementa el algoritmo, naturalmente es miembro de la clase `ListaDoble`. El primer argumento, `anterior`, representa un puntero al nodo `n` a partir del cual se enlaza el nuevo. El segundo argumento, `entrada`, es el dato que se añade a la lista.

```
void ListaDoble::insertaDespues(NodoDoble* anterior, Dato entrada)
{
  NodoDoble* nuevo;

  nuevo = new NodoDoble(entrada);
  nuevo -> ponerAdelante(anterior -> adelanteNodo());
  if (anterior -> adelanteNodo() != NULL)
    anterior -> adelanteNodo() -> ponerAtras(nuevo);
  anterior-> ponerAdelante(nuevo);
  nuevo -> ponerAtras(anterior);
}
```

www.FreeLibros.me

10.8.3. Eliminar un nodo de una lista doblemente enlazada

Quitar un nodo de una lista doble supone ajustar los enlaces de dos nodos, el nodo *anterior* con el nodo *siguiente* al que se desea eliminar. El puntero `adelante` del nodo anterior debe apuntar al nodo *siguiente*, y el puntero `atras` del nodo *siguiente* debe apuntar al nodo *anterior.*

El algoritmo es similar al del borrado para una lista simple, más simple, ya que ahora la dirección del nodo *anterior* se encuentra en el campo `atras` del nodo a borrar. Los pasos a seguir:

1. Búsqueda del nodo que contiene el dato.
2. El puntero `adelante` del nodo anterior tiene que apuntar al puntero `adelante` del nodo a eliminar (si no es el nodo *cabecera*).
3. El puntero `atras` del nodo siguiente a borrar tiene que apuntar a donde apunta el puntero `atras` del nodo a eliminar (si no es el último nodo).
4. Si el nodo que se elimina es el primero, se modifica `cabeza` para que tenga la dirección del nodo `siguiente`.
5. La memoria ocupada por el nodo es liberada.

La implementación del algoritmo es una función miembro de la clase `ListaDoble`.

```
void ListaDoble::eliminar (Dato entrada)
{
  NodoDoble* actual;
  bool encontrado = false;

  actual = cabeza;
  // Bucle de búsqueda
  while ((actual != NULL) && (!encontrado))
  {
    encontrado = (actual -> datoNodo() == entrada);
    if (!encontrado)
      actual = actual -> adelanteNodo();
  }
  // Enlace de nodo anterior con el siguiente
  if (actual != NULL)
  {
    //distingue entre nodo cabecera o resto de la lista
    if (actual == cabeza)
    {
      cabeza = actual -> adelanteNodo();
      if (actual -> adelanteNodo() != NULL)
        actual -> adelanteNodo() -> ponerAtras(NULL);
    }
    else if (actual -> adelanteNodo() != NULL) // No es el último
    {
      actual->atrasNodo()->ponerAdelante(actual->adelanteNodo());
      actual->adelanteNodo()->ponerAtras(actual->atrasNodo());
    }
    else      // último nodo
      actual->atrasNodo()->ponerAdelante(NULL);
  }
}
```

www.FreeLibros.me

EJERCICIO 10.2. Se crea una lista doblemente enlazada con números enteros, del 1 al 999 generados aleatoriamente. Una vez creada la lista se eliminan los nodos que estén fuera de un rango de valores leídos desde el teclado.

En el Apartado 10.8 se han declarado las clases `NodoDoble` y `ListaDoble` necesarias para realizar este ejercicio. Se añade la función `visualizar()` para recorrer la lista doble y mostrar por pantalla los datos de los nodos. Además, se declara la clase `IteradorLista`, para *visitar* cada nodo de cualquier lista doble (de enteros); en esta ocasión se utiliza el mecanismo friend para que `IteradorLista` pueda acceder a los miembros protegidos de `ListaDoble`. El constructor de `IteradorLista` asocia la lista a recorrer con el objeto iterador. En la clase iterador se implementa la función `siguiente()`, cada llamada a `siguiente()`devuelve el puntero al nodo actual de la lista y avanza al siguiente nodo. Una vez visitados todos los nodos de la lista devuelve `NULL`.

La función `main()` genera los números aleatorios, que se insertan en la lista doble. A continuación, se pide el rango de elementos a eliminar; con el objeto iterador se obtienen los nodos y aquéllos fuera de rango se borran de la lista.

```
// archivo con la clase NodoDoble
#include "NodoDoble.h"

// declaración friend en la clase ListaDoble
class IteradorLista;
class ListaDoble
{
  friend class IteradorLista;
    // ...
};
void ListaDoble::visualizar()
{
  NodoDoble* n;
  int k = 0;
  n = cabeza;
  while (n != NULL)
  {
    char c;
    k++; c = (k % 15 != 0 ? ' ' : '\n');
    cout << n -> datoNodo() << c;
    n = n -> adelanteNodo();
  }
}
// archivo con la clase IteradorLista
class IteradorLista
{
protected:
    NodoDoble* actual;
public:
    IteradorLista(ListaDoble& ld)
    {
      actual = ld.cabeza;
    }
    NodoDoble* siguiente()
    {
```

www.FreeLibros.me

```
      NodoDoble* a;
      a = actual;
      if (actual != NULL)
      {
        actual = actual -> adelanteNodo();
      }
      return a;
    }
  };

/*
   función main(). Crea el objeto lista doble e inserta
   datos enteros generados aleatoriamente.
   Crea objeto iterador de lista, para recorrer sus elementos y
   aquellos fuera de rango se eliminan. El rango se lee del teclado.
*/

#include <stdlib.h>
#include <time.h>
#define N 999
#define randomize (srand(time(NULL)))
#define random(num) (rand()%(num))
#include <iostream>
#include "NodoDoble.h"
#include "ListaDoble.h"
#include "IteradorLista.h"
using namespace std;
typedef int Dato;

int main()
{
  int d, x1, x2, m;

  ListaDoble listaDb;
  cout << "Número de elementos de la lista: ";
  cin >> m;
  for (int j = 1; j <= m; j++)
  {
    d = random(N) + 1;
    listaDb.insertarCabezaLista(d);
  }

  cout << "Elementos de la lista original" << endl;
  listaDb.visualizar();
  // rango de valores
  cout << "\nRango que va a contener la lista: " << endl;
  cin >> x1 >> x2;
  // recorre la lista con el iterador
  IteradorLista *iterador;
  iterador = new IteradorLista(listaDb);
  NodoDoble* a;

  a = iterador -> siguiente();
  while (a != NULL)
```

www.FreeLibros.me

```
    {
      int w;
      w = a -> datoNodo();
      if (!(w >= x1 && w <= x2))
        // fuera de rango
        listaDb.eliminar(w);
      a = iterador -> siguiente();
    }

    // muestra los elementos de la lista
    cout << "Elementos actuales de la lista" << endl;
    listaDb.visualizar();
    return 0;
  }
```

10.9. LISTAS CIRCULARES

En las listas lineales simples o en las dobles siempre hay un primer nodo (*cabeza*) y un último nodo (*cola*). Una lista circular, por propia naturaleza, no tiene ni principio ni fin. Sin embargo, resulta útil establecer un nodo de acceso a la lista y, a partir de él, al resto de sus nodos.

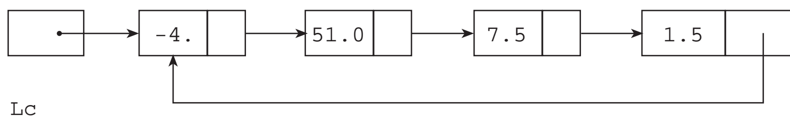


Figura 10.14. Lista circular.

Las operaciones que se realizan sobre una lista circular son similares a las operaciones sobre listas lineales, teniendo en cuenta que no hay *primero* ni `último` nodo, aunque sí un nodo de acceso. Son las siguientes:

- *Inicialización.*
- *Inserción de elementos.*
- *Eliminación de elementos.*
- *Búsqueda de elementos.*
- *Recorrido de la lista circular.*
- *Verificación de lista vacía.*

Nota de programación

Una lista circular es un tipo abstr acto de datos f ormado por elementos de cualquier tipo y unas operaciones características. En C++ se implementa con la clase *ListaCircular*.

www.FreeLibros.me

10.9.1. Implementación de la clase ListaCircular

La construcción de una lista circular se puede hacer con enlace simple o enlace doble entre sus nodos. A continuación se implementa utilizando un enlace simple.

La clase `ListaCircular` dispone del puntero de acceso a la lista, junto a las funciones que implementan las operaciones.

La creacción de un nodo varía respecto al de las listas no circulares, el campo `enlace`, en vez de inicializarse a `NULL`, se inicializa para que apunte a sí mismo, de tal forma que es una lista circular de un solo nodo. La funcionalidad (la *interfaz*) de la clase `NodoCircular` es la misma que la de un `Nodo` de una lista enlazada.

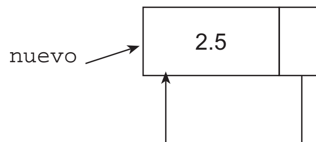


Figura 10.15. Creacción de un nodo en lista circular.

```
// archivo de cabecera NodoCircular.h
typedef int Dato;
class NodoCircular
{
private:
    Dato dato;
    NodoCircular* enlace;
public:
    NodoCircular (Dato entrada)
    {
      dato = entrada;
      enlace = this;    // se apunta a sí mismo
    }
      // ...
};
```

La clase `ListaCircular`:

```
class ListaCircular
{
  private:
    NodoCircular* lc;
  public:
    ListaCircular()
    {
      lc = NULL;
    }
    void insertar(Dato entrada);
    voideliminar(Dato entrada);
    void recorrer();
    void borrarLista();
    NodoCircular* buscar(Dato entrada);
};
```

www.FreeLibros.me

10.9.2. Insertar un elemento

El algoritmo empleado para añadir o insertar un elemento en una lista circular varía dependiendo de la posición en que se desea insertar. La implementación realizada considera que `lc` tiene la dirección del último nodo, e inserta un nodo en la posición anterior a `lc`.

```
void ListaCircular::insertar(Dato entrada)
{
  NodoCircular* nuevo;
  nuevo = new NodoCircular(entrada);
  if (lc != NULL)     // lista circular no vacía
  {
    nuevo -> ponerEnlace(lc -> enlaceNodo());
    lc -> ponerEnlace(nuevo);
  }
  lc = nuevo;
}
```

10.9.3. Eliminar un elemento

Eliminar un nodo de una lista circular sigue los mismos pasos que los dados para eliminar un nodo en una lista lineal. Hay que enlazar el nodo anterior con el siguiente al que se desea eliminar y liberar la memoria que ocupa. El algoritmo es el siguiente:

1. Búsqueda del nodo.
2. Enlace del nodo anterior con el siguiente.
3. En caso de que el nodo a eliminar sea el de acceso de a la lista, `lc`, se modifica *lc* para que tenga la dirección del nodo anterior.
4. Por último, liberar la memoria ocupada por el nodo.

La implementación debe de tener en cuenta que la lista circular conste de un solo nodo, ya que al eliminarlo la lista se queda vacía. La condición `lc == lc` → `enlaceNodo()` determina si la lista consta de un solo nodo.

La función recorre la lista buscando el nodo con el dato a eliminar, utiliza un puntero al nodo *anterior* para que cuando encuentre el nodo se enlace con el *siguiente*. Se accede al dato del nodo con la sentencia: `actual`→ `enlaceNodo()`→ `datoNodo()`, de tal forma que si coincide con el dato a eliminar, en `actual` está la dirección el nodo anterior. Después del bucle se vuelve a preguntar por el campo dato, ya que no se comparó el nodo de acceso a la lista y el bucle puede terminar sin encontrar el nodo.

```
void ListaCircular:: eliminar (Dato entrada)
{
  NodoCircular* actual;
  bool encontrado = false;

  actual = lc;
  while ((actual -> enlaceNodo() != lc) && (!encontrado))
  {
    encontrado = (actual->enlaceNodo()->datoNodo() == entrada);
    if (!encontrado)
```

www.FreeLibros.me

```
      {
        actual = actual -> enlaceNodo();
      }
    }
    encontrado = (actual->enlaceNodo()->datoNodo() == entrada);
    // Enlace de nodo anterior con el siguiente
    if (encontrado)
    {
      NodoCircular* p;
      p = actual -> enlaceNodo();      // Nodo a eliminar
      if (lc == lc -> enlaceNodo())    // Lista de un nodo
       lc = NULL;
      else
      {
        if (p == lc)
            lc = actual; // el nuevo acceso a la lista es el anterior
        actual -> ponerEnlace(p -> enlaceNodo());
      }
      delete p;
    }
  }
```

10.9.4. Recorrer una lista circular

Una operación común de todas las estructuras enlazadas es recorrer o visitar todos los nodos de la estructura. En una lista circular el recorrido puede empezar en cualquier nodo, a partir de uno dado procesa cada nodo hasta alcanzar el nodo de partida. La función, miembro de la clase `ListaCircular`, inicia el recorrido en el nodo siguiente al de acceso a la lista, `lc`, y termina cuando alcanza de nuevo al nodo `lc`. El proceso que se realiza con cada nodo consiste en escribir su contenido.

```
void ListaCircular:: recorrer()
{
  NodoCircular* p;
  if (lc != NULL)
  {
    p = lc -> enlaceNodo();  // siguiente nodo al de acceso
    do {
      cout << "\t" << p -> datoNodo();
      p = p -> enlaceNodo();
    }while(p != lc -> enlaceNodo());
  }
  else
     cout << "\t Lista Circular vacía." << endl;
}
```

EJERCICIO 10.3. Crear una lista circular con palabras leídas del teclado. El programa debe presentar estas opciones:

- *Mostrar las cadenas que forman la lista.*
- *Borrar una palabra dada.*
- *Al terminar la ejecución, recorrer la lista eliminando los nodos.*

www.FreeLibros.me

El atributo dato del nodo de la lista es de tipo `string`. Las cadenas se leen del teclado con la función `getline()`, cada cadena leída se inserta en la lista circular, llamando a la función `insertar()` de la clase `ListaCircular`.

La función `eliminar()` busca el nodo que tiene una palabra y le retira de la lista. La comparación de cadenas, tipo `string`, se puede realizar con el operador == (está sobrecargado). Con el fin de recorrer la lista circular liberando cada nodo, se implementa `borrarLista()` de la clase `ListaCircular`.

```
void ListaCircular::borrarLista()
{
  NodoCircular* p;
  if (lc != NULL)
  {
    p = lc;
    do {
      NodoCircular* t;
      t = p;
      p = p -> enlaceNodo();
      delete t;       // no es estrictamente necesario
     }while(p != lc);
  }
  else
    cout << "\n\t Lista vacía." << endl;
  lc = NULL;
}

/*
   función main(): escribe un sencillo menu para
   elegir operaciones con la lista circular.
*/

#include <iostream>
using namespace std;
typedef string Dato;
#include "NodoCircular.h"
#include "ListaCircular.h"

int main()
  {
  ListaCircular listaCp;
  int opc;
  char palabra[81];

  cout << "\n Entrada de Nombres. Termina con FIN\n";
  do
  {
  cin.getline(palabra,80);
  if (strcmp(palabra,"FIN") != 0)
  listaCp.insertar(palabra);
  }while (strcmp(palabra,"FIN")!=0);

  cout << "\t\tLista circular de palabras" << endl;
  listaCp.recorrer();
```

www.FreeLibros.me

```
    cout << "\n\t Opciones para manejar la lista" << endl;
    do {
      cout << "1. Eliminar una palabra.\n";
      cout << "2. Mostrar la lista completa.\n";
      cout << "3. Salir y eliminar la lista.\n";
      do {
        cin >> opc;
      }while (opc < 1 || opc > 3);

      switch (opc) {
      case 1: cout << "Palabra a eliminar: ";
              cin.ignore();
              cin.getline(palabra,80);
              cin.ignore();
              listaCp.eliminar(palabra);
              break;
      case 2: cout << "Palabras en la Lista:\n";
              listaCp.recorrer(); cout << endl;
              break;
      case 3: cout << "Eliminación de la lista." << endl;
              listaCp.borrarLista();
     }
  }while (opc != 3);
  return 0;
}
```

10.10. LISTAS ENLAZADAS GENÉRICAS

La definición de una lista está muy ligada al tipo de datos de sus elementos; así, se han puesto ejemplos en los que el tipo es `int`, otros, en los que el tipo es `double`, otros, `string`. C++ dispone del mecanismo `template` para declarar clases y funciones con independencia del tipo de dato de al menos un elemento. Entonces, el tipo de los elementos de una lista gerérica será *un tipo genérico T*, que será conocido en el momento de crear la lista.

```
template <class T> class ListaGenerica
template <class T> class NodoGenerico

ListaGenerica<double> lista1;   // lista de número reales
ListaGenerica<string> lista2;   // lista de cadenas
```

10.10.1. Declaración de la clase `ListaGenérica`

Las operaciones del tipo *lista genérica* son las especificadas en el Apartado 10.2. En el Apartado 10.3 se delaró la clase `NodoGenerico`, ahora se declara y se implementa la clase `ListaGenerica`.

```
// archivo ListaGenerica.h

template <class T> class ListaGenerica
{
```

www.FreeLibros.me

```
protected:
  NodoGenerico<T>* primero;

public:
  ListaGenerica(){ primero = NULL;}
  NodoGenerico<T>* leerPrimero() const { return primero;}

  void insertarCabezaLista(T entrada);
  void insertarUltimo(T entrada);
  void insertarLista(NodoGenerico<T>* anterior, T entrada);
  NodoGenerico<T>* ultimo();
  void eliminar(T entrada);
  NodoGenerico<T>* buscarLista(T destino);
 };
```

A continuación, la implementación de las funciones miembro, que también son funciones genéricas.

```
// inserción por la cabeza de la lista
template <class T>
void ListaGenerica<T>::insertarCabezaLista(T entrada)
{
  NodoGenerico<T>* nuevo;
  nuevo = new NodoGenerico<T>(entrada);
  nuevo -> ponerEnlace(primero);// enlaza nuevo con primero
  primero = nuevo; // mueve primero y apunta al nuevo nodo
}
// inserción por la cola de la lista
template <class T>
void ListaGenerica<T>::insertarUltimo(T entrada)
{
  NodoGenerico<T>* ultimo = this -> ultimo();
  ultimo -> ponerEnlace(new NodoGenerico<T>(entrada));
}
// recorre hasta el último nodo la lista
template <class T>
NodoGenerico<T>* ListaGenerica<T>::ultimo()
{
  NodoGenerico<T>* p = primero;
  if (p == NULL ) throw "Error, lista vacía";
  while (p -> enlaceNodo()!= NULL) p = p -> enlaceNodo();
  return p;
}
// inserción entre dos nodos de la lista
template <class T> void
ListaGenerica<T>::insertarLista(NodoGenerico<T>* ant, T entrada)
{
  NodoGenerico<T>* nuevo = new NodoGenerico<T>(entrada);
  nuevo -> ponerEnlace(ant -> enlaceNodo());
  ant -> ponerEnlace(nuevo);
}
// búsqueda, si el elemento correspondiente a T es una clase
// debe redefinir el operador de comparación ==
template <class T>
NodoGenerico<T>* ListaGenerica<T>::buscarLista(T destino)
```

www.FreeLibros.me

```
{
  NodoGenerico<T>* indice;
  for (indice = primero; indice!= NULL; indice = indice->enlaceNodo())
    if (destino == indice -> datoNodo())
      return indice;
  return NULL;
}
// borra el primer nodo encontrado con dato
template <class T>
void ListaGenerica<T>:: eliminar(T entrada)
{
  NodoGenerico<T> *actual, *anterior;
  bool encontrado;
  actual = primero;
  anterior = NULL;
  encontrado = false;
// búsqueda del nodo y del anterior
  while ((actual != NULL) && !encontrado)
  {
    encontrado = (actual -> datoNodo() == entrada);
    if (!encontrado)
    {
      anterior = actual;
      actual = actual -> enlaceNodo();
    }
  }
  // enlace del nodo anterior con el siguiente
  if (actual != NULL)
  {
    // Distingue entre cabecera y resto de la lista
    if (acual == primero)
    {
      primero = actual -> enlaceNodo();
    }
    else
      anterior -> ponerEnlace(actual -> enlaceNodo());
    delete actual;
    }
}
```

10.10.2. Iterador de `ListaGenerica`

Un objeto `Iterador` se diseña para recorrer los elementos de un contenedor. Un iterador de una lista enlazada accede a cada uno de sus nodos de la lista, hasta alcanzar el último elemento. El constructor del objeto iterador inicializa el puntero `actual` al primer elemento de la estructura; la función `siguiente()` devuelve el elemento `actual` y hace que éste quede apuntando al siguiente elemento. Si no hay siguiente, devuelve `NULL`.

La clase `ListaIterador` implementa el iterador de una lista enlazada genérica y, en consecuencia, también será clase genérica.

```
template <class T> class ListaIterador
{
```

www.FreeLibros.me

```
private:
  NodoGenerico<T> *prm, *actual;
public:
  ListaIterador(const ListaGenerica<T>& list)
  {
    prm = actual = list.leerPrimero();
  }

  NodoGenerico<T> *siguiente()
  {
    NodoGenerico<T> * s;
    if (actual != NULL)
    {
      s = actual;
      actual = actual -> enlaceNodo();
    }
    return actual;
  }
  void iniciaIterador()     // pone en actual la cabeza de la lista
  {
    actual = prm;
  }
}
```

RESUMEN

Una **lista enlazada** es una estructura de datos dinámica, que se crea vacía y crece o decrece en tiempo de ejecución. Los componentes de una lista están ordenados por sus campos de enlace en vez de ordenados físicamente como están en un array. El final de la lista se señala mediante una constante o puntero especial llamado `NULL`. La principal ventaja de una lista enlazada sobre un array radica en el tamaño dinámico de la lista, ajustándose al número de elementos. Por contra, la desventaja de la lista frente a un array está en el acceso a los elementos, para un array el acceso es directo, a partir del índice, para la lista el acceso a un elemento se realiza mediante el campo enlace entre nodos.

Una **lista simplemente enlazada** contiene sólo un enlace a un sucesor único, a menos que sea el último, en cuyo caso no se enlaza con ningún otro nodo.

Cuando se inserta un elemento en una lista enlazada, se deben considerar cuatro casos: añadir a una lista vacía, añadir al principio de la lista, añadir en el interior y añadir al final de la lista.

Para borrar un elemento, primero hay que buscar el nodo que lo contiene y considerar dos casos: borrar el primer nodo y borrar cualquier otro nodo de la lista.

El recorrido de una lista enlazada significa pasar por cada nodo (*visitar*) y procesarlo. El proceso de cada nodo puede consistir en escribir su contenido, modificar el campo dato, ...

Una **lista doblemente enlazada** es aquélla en la que cada nodo tiene una referencia a su sucesor y otra a su predecesor. Las listas doblemente enlazadas se pueden recorrer en ambos sentidos. Las operaciones básicas son inserción, borrado y recorrer la lista; similares a las listas simples.

Una **lista enlazada circularmente** por propia naturaleza no tiene primero ni último nodo. Las listas circulares pueden ser de enlace simple o doble.

Una **lista enlazada genérica** tiene como tipo de dato el *tipo genérico T*, es decir, el tipo concreto se especificará en el momento de crear el objeto lista. La construcción de una *lista genérica* se realiza con las plantillas (`template`), mediante dos clases genéricas: `NodoGenerico` y `ListaGenerica`.

www.FreeLibros.me