Capítulo 10
Listas

Objetivos

Con el estudio de este capítulo usted podrá:

- Distinguir entre una estructura secuencial y una estructura enlazada.
- Definir el tipo abstracto de datos *Lista*.
- Conocer las operaciones básicas de la listas enlazadas.
- Implementar una lista enlazada para un tipo de elemento.
- Aplicar la estructura Lista para almacenar datos en el desarrollo de aplicaciones.
- Definir una lista doblemente enlazada.
- Definir una lista circular.
- Implementar listas enlazadas ordenadas.
- Realizar una lista genérica.

Contenido

Conceptos clave

- Enlace.
- Estructura enlazada.
- Lista circular.
- Lista doble.
- Lista doblemente enlazada.
- Lista genérica.
- Lista simplemente enlazada.
- Nodo.
- Recorrer una lista
- Tipo abstracto de dato.

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.

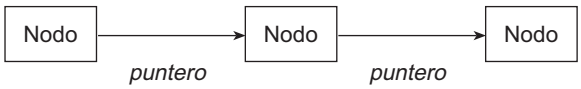
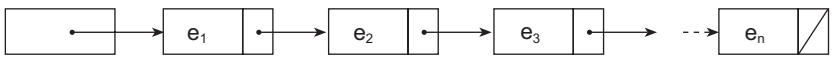


Figura 10.1. Lista enlazada (representación simple).

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).

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.

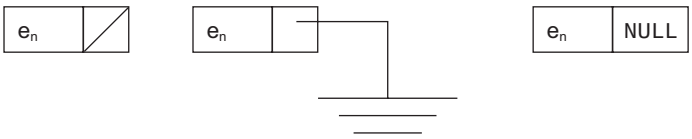


Figura 10.3. Diferentes representaciones gráficas del nodo último.

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.

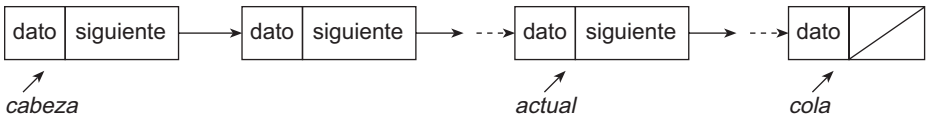


Figura 10.4. Representación gráfica de una lista enlazada.

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`.

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
```

```
  {
    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;
    }
};
```

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.

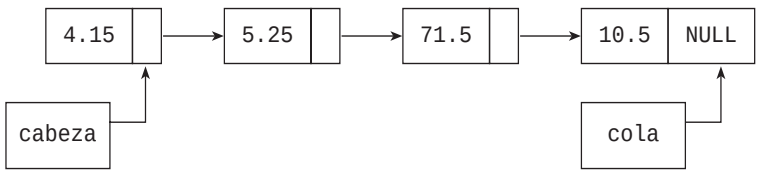


Figura 10.5. Declaraciones de tipos en lista enlazada.

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:

- 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`).

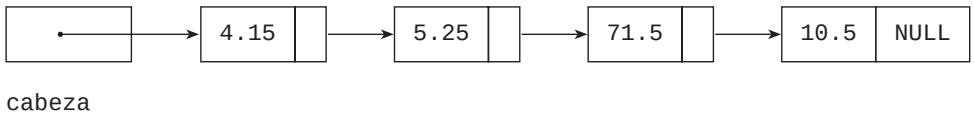


Figura 10.6. Acceso a una lista con puntero cabeza.

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).

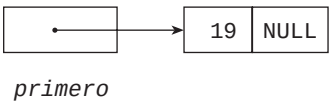
```
typedef int Dato;
#include "Nodo.h"
```

```
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*.

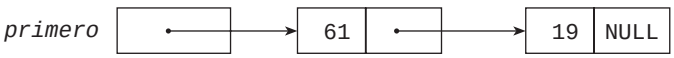
A continuación, y antes de escribir su código, se muestra el comportamiento de la función `crearLista()`. En primer lugar, se crea un nodo con un valor y su dirección se asigna a `primero`:

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



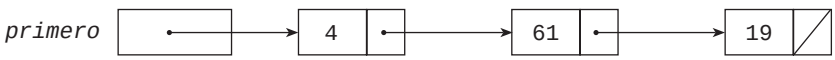
Ahora se desea añadir un nuevo elemento con el valor `61`, y situarlo en el primer lugar de la lista. Se utiliza el constructor de `Nodo` que enlaza con otro nodo ya creado:

```
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)
```

```
        {
          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.

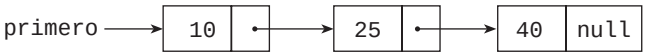
10.4.1. Insertar en la cabeza de la lista

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:

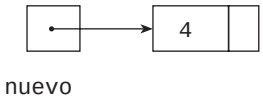
1. Crear un nodo e inicializar el campo dato al nuevo elemento. La dirección del nodo creado se asigna a `nuevo`.
2. Hacer que el campo `enlace` del nodo creado apunte a la *cabeza* (`primero`) de la lista.
3. Hacer que `primero` apunte al nodo que se ha creado.

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

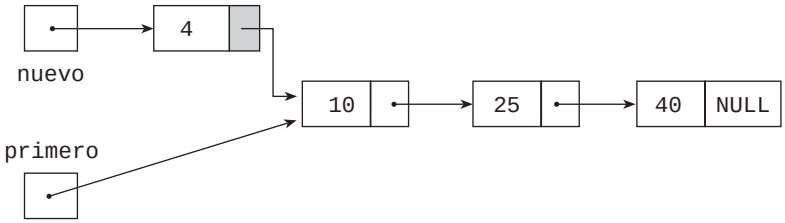
EJEMPLO 10.3. Una lista enlazada contiene tres elementos , 10, 25 y 40. Insertar un nuevo elemento, 4, en cabeza de la lista.



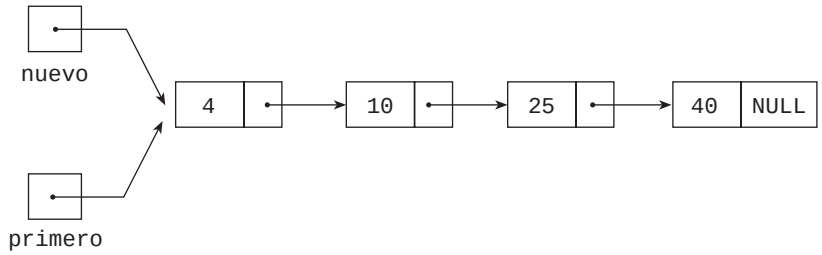
Paso 1



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. Una vez creada la lista, se ha de recorrer para mostrar los números 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

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;
}
```

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`.

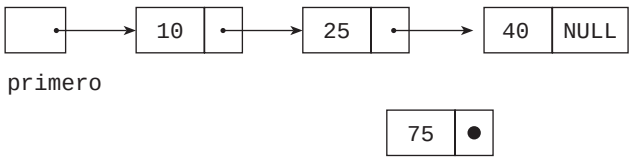


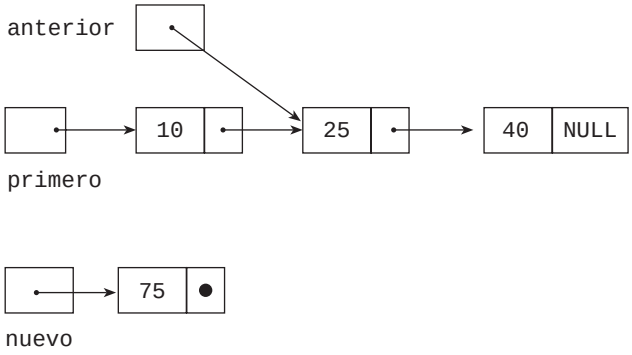
Figura 10.7. Inserción entre dos nodos.

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

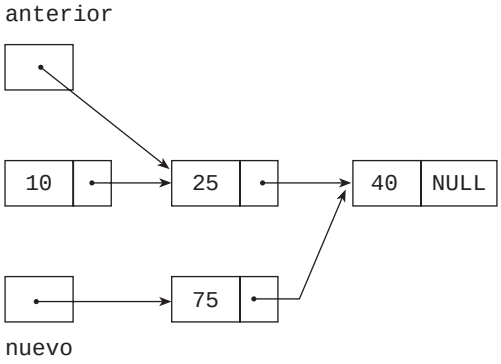
1. Crear un nodo con el elemento y el campo `enlace` a `NULL`.
2. Poner campo `enlace` del nuevo nodo apuntando a `n2`, ya que el nodo creado se ubicará justo antes de `n2`.
3. Si el puntero `anterior` tiene la dirección del nodo `n1`, entonces poner su atributo `enlace` apuntando al nodo creado.

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

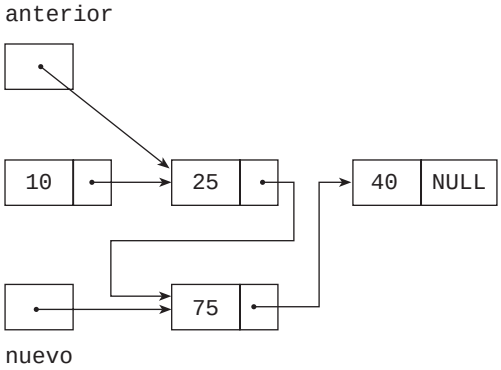
Etapas 1



Etapa 2



Etapa 3



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

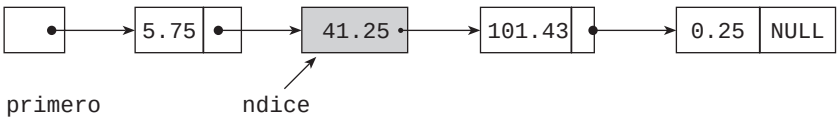


Figura 10.8. Búsqueda en una lista.

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;
}
```

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;
```

```
  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.

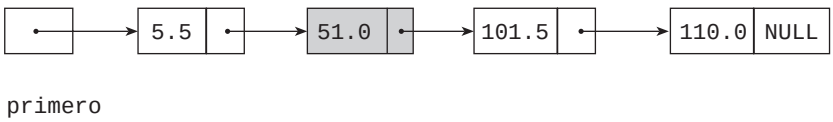


Figura 10.9. Lista ordenada.

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.

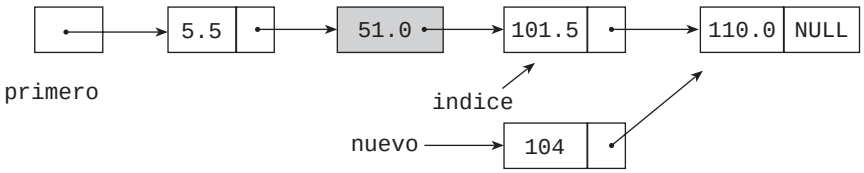


Figura 10.10. Inserción en una lista ordenada.

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());
```

```
      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.

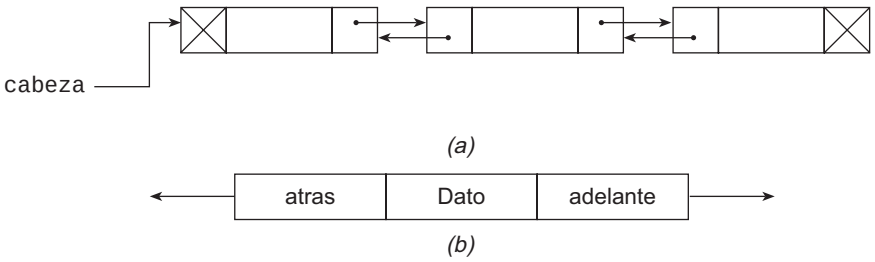


Figura 10.11. Lista doblemente enlazada. *(a)* Lista con tres nodos; *(b)* nodo.

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.

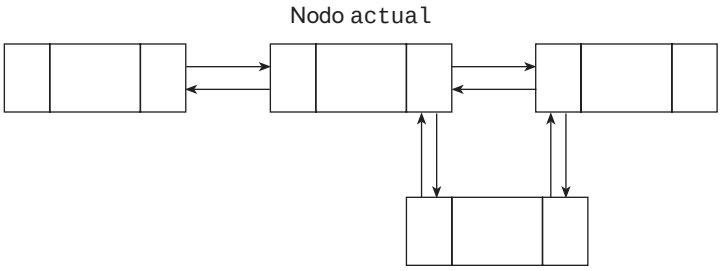


Figura 10.12. Inserción de un nodo en una lista dob lemente enlazada.

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.

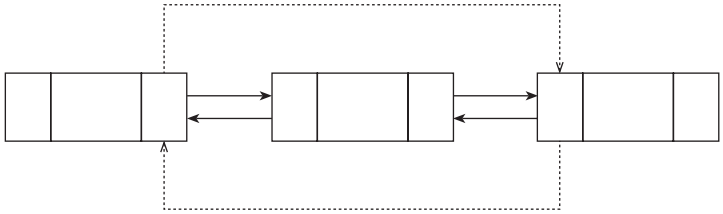


Figura 10.13. Eliminación de un nodo en una lista doblemente enlazada.

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.

- 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);
}
```

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);
  }
}
```

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()
    {
```

```
      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)
```

```
  {
    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.

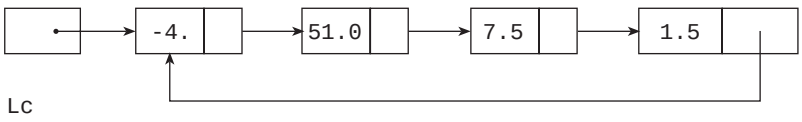


Figura 10.14. Lista circular.

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 formado por elementos de cualquier tipo y unas operaciones características. En C++ se implementa con la clase *ListaCircular*.

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.

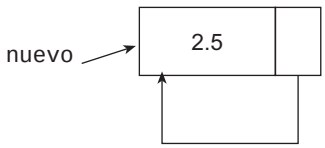


Figura 10.15. Creacción de un nodo en lista circular.

```
// 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);
};
```

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)
```

```
    {
      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.*

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();
```

```
      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
{
```

```
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)
```

```
{
  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
{
```

```
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`.

EJERCICIOS

10.1. Escribir un función, en la clase `Lista`, que devuelva cierto si la lista está vacía.

10.2. Añadir a la clase `ListaDoble` un función que devuelva el número de nodos de una lista doble.

10.3. En una lista enlazada de números enteros se desea añadir un nodo entre dos nodos consecutivos cuyos datos tienen distinto signo. El nuevo nodo debe ser la diferencia en valor absoluto de los dos nodos.

10.4. A la clase *Lista* añadir la función `eliminaraPosicion()` que retire el nodo que ocupa la posición i, siendo 0 la posición del nodo cabecera.

10.5. Escribir un función que tenga como argumento el puntero al primer nodo de una lista enlazada, y cree una lista doblemente enlazada con los mismos atributos dato pero en orden inverso.

10.6. Se tiene una lista simplemente enlazada de números reales. Escribir una función para obtener una lista doble ordenada respecto al atributo dato, con los valores reales de la lista simple.

10.7. Escribir una función para crear una lista doblemente enlazada de palabras introducidas por teclado. El acceso a la lista debe ser el nodo que está en la posición intermedia.

10.8. La clase `ListaCircular` dispone de las funciones que implementan las operaciones de una lista circular de palabras. Escribir un función miembro que cuente el número de veces que una palabra dada se encuentra en la lista.

10.9. Escribir una función que devuelva el mayor entero de una lista enlazada de números enteros.

10.10. Se tiene una lista de simple enlace, el campo dato son objetos *Alumno* con las variables: *nombre, edad, sexo*. Escribir una función para transformar la lista de tal forma que si el primer nodo es de un alumno de sexo masculino el siguiente sea de sexo femenino, así alternativamente, siempre que sea posible, masculino y femenino.

10.11. Supóngase una lista circular de cadenas ordenada alfabéticamente. El puntero de acceso a la lista tiene la dirección del nodo alfabéticamente mayor. Escribir un función para añadir una nueva palabra, en el orden que le corresponda, a la lista.

10.12. Dada la lista del Ejercicio 10.11 escribir un función que elimine una palabra dada.

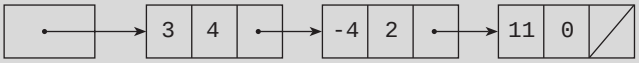
PROBLEMAS

10.1. Escribir un programa que realicen las siguientes tareas:

- Crear una lista enlazada de números enteros positivos al azar. Insertar por el último nodo.
- Recorrer la lista para mostrar los elementos por pantalla.
- Eliminar todos los nodos que superen un valor dado.

10.2. Se tiene un archivo de texto de palabras separadas por un blanco o el carácter fin de línea. Escribir un programa para formar una lista enlazada con las palabras del archivo. Una vez formada la lista, añadir nuevas palabras o borrar alguna de ellas. Al finalizar el programa escribir las palabras de la lista en el archivo.

10.3. Un polinomio se puede representar como una lista enlazada. El primer nodo representa el primer término del polinomio, el segundo nodo al segundo término del polinomio y así sucesivamente. Cada nodo tiene como campo dato el coeficiente del término y su exponente. Por ejemplo, $3x^4 - 4x^2 + 11$ se representa:



Escribir un programa que dé entrada a polinomios en x, los represente en una lista enlazada simple. A continuación, obtenga valores del polinomio para valores de `x = 0.0, 0.5, 1.0, 1.5, … , 5.0`

10.4. Teniendo en cuenta la representación de un polinomio propuesta en el Problema 10.3, hacer los cambios necesarios para que la lista enlazada sea circular. La referencia de acceso debe tener la dirección del último término del polinomio, el cuál apuntará al primer término.

10.5. Según la representación de un polinomio propuesta en el Problema 10.4, escribir un programa para realizar las siguientes operaciones:

- Obtener la lista circular suma de dos polinomios.
- Obtener el polinomio derivada.
- Obtener una lista circular que sea el producto de dos polinomios.

10.6. Escribir un programa para obtener una lista doblemente enlazada con los caracteres de una cadena leída desde el teclado. Cada nodo de la lista tendrá un carácter. Una vez que se haya creado la lista, ordenarla alfabéticamente y escribirla por pantalla.

10.7. Un conjunto es una secuencia de elementos todos del mismo sin duplicados. Escribir un programa para representar un conjunto de enteros con una lista enlazada. El programa debe contemplar las operaciones:

- Cardinal del conjunto.
- Pertenencia de un elemento al conjunto.
- Añadir un elemento al conjunto.
- Escribir en pantalla los elementos del conjunto.

10.8. Con la representación propuesta en el Problema 10.7, añadir las operaciones básicas de conjuntos:

- Unión de dos conjuntos.
- Intersección de dos conjuntos.
- Diferencia de dos conjuntos.
- Inclusión de un conjunto en otro.

10.9. Escribir un programa en el que dados dos archivos *F1*, *F2* formados por palabras separadas por un blanco o fin de línea, se creen dos conjuntos con las palabras de *F1* y *F2* respectivamente. Posteriormente, encontrar las palabras comunes y mostrarlas por pantalla.

10.10. Utilizar una lista doblemente enlazada para controlar una lista de pasajeros de una línea aérea. El programa principal debe ser controlado por menú y permitir al usuario visualizar los datos de un pasajero determinado, insertar un nodo (siempre por el final), eliminar un pasajero de la lista. A la lista se accede por dos variables, una referencia al primer nodo y la otra al último nodo.

10.11. Para representar un entero largo, de más de 30 dígitos, utilizar una lista circular cuyos nodos tienen como atributo dato un dígito del entero largo. Escribir un programa cuya entrada sea dos enteros largos y se obtenga su suma.

10.12. Un vector disperso es aquel que tiene muchos elementos que son cero. Escribir un programa que represente un vector disperso con listas enlazadas. Los nodos son los elementos del vector distintos de cero. Cada nodo contendrá el valor del elemento y su índice (posición del vector). El programa ha de realizar las operaciones: sumar dos vectores de igual dimensión y hallar el producto escalar.

Capítulo 11
Pilas

Objetivos

Con el estudio de este capítulo usted podrá:

- Especificar el tipo abstracto de datos Pila.
- Conocer aplicaciones de las Pilas en la programación.
- Definir e implementar la clase Pila.
- Conocer las diferentes formas de escribir una expresión.
- Evaluar una expresión algebraica.

Contenido

Conceptos clave

- Concepto de tipo abstracto de datos.
- Concepto de una pila.
- Expresiones y sus tipos.
- Listas enlazadas.
- Notación de una expresión.
- Prioridad.

INTRODUCCIÓN

En este capítulo se estudian en detalle la estructura de datos *Pila* utilizada frecuentemente en la resolución de algoritmos. La *Pila* es una estructura de datos que almacena y recupera sus elementos atendiendo a un estricto orden. Las pilas se conocen también como estructuras **LIFO** (*Last-in, First-Out, último en entrar primero en salir*). El desarrollo de las pilas como tipos abstractos de datos es el motivo central de este capítulo.

Las pilas se utilizan en compiladores, sistemas operativos y programas de aplicaciones. Una aplicación interesante es la evaluación de expresiones algebraicas mediante pilas.

11.1. CONCEPTO DE PILA

Una **pila** (*stack*) es una colección ordenada de elementos a los que sólo se puede acceder por un único lugar o extremo. Los elementos de la pila se añaden o quitan (borran) de la misma sólo por su parte superior, la **cima** de la pila. Éste es el caso de una pila de platos, una pila de libros, etc.

Definición

Una pila es una estructura de datos de entradas ordenadas tales que sólo se pueden introducir y eliminar por un extremo, llamado cima.

Cuando se dice que la pila está ordenada, se quiere decir que hay un elemento al que se puede acceder primero (el que está encima de la pila), otro elemento al que se puede acceder en segundo lugar (justo el elemento que está debajo de la cima), un tercero, etc. No se requiere que las entradas se puedan comparar utilizando el operador "*menor que*" y pueden ser de cualquier tipo.

Las entradas de la pila deben ser eliminadas en el orden inverso al que se situaron en la misma. Por ejemplo, se puede crear una pila de libros, situando primero un diccionario, encima de él una enciclopedia y encima de ambos una novela de modo que la pila tendrá la novela en la parte superior.

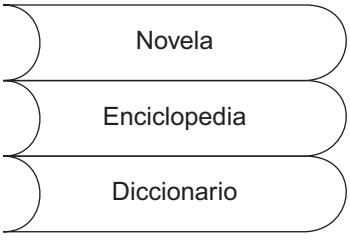


Figura 11.1. Pila de libros.

Cuando se quitan los libros de la pila, primero debe quitarse la novela, luego la enciclopedia y por último el diccionario.

Debido a su propiedad específica "*último en entrar, primero en salir*" se conoce a las pilas como estructura de datos **LIFO** (*last-in, first-out*).

Las operaciones usuales en la pila son *Insertar* y *Quitar*. La operación **`Insertar`** (`push`) añade un elemento en la cima de la pila y la operación **`Quitar`** (pop) elimina o saca un elemento de la pila. La Figura 11.2 muestra una secuencia de operaciones *Insertar* y *Quitar*.

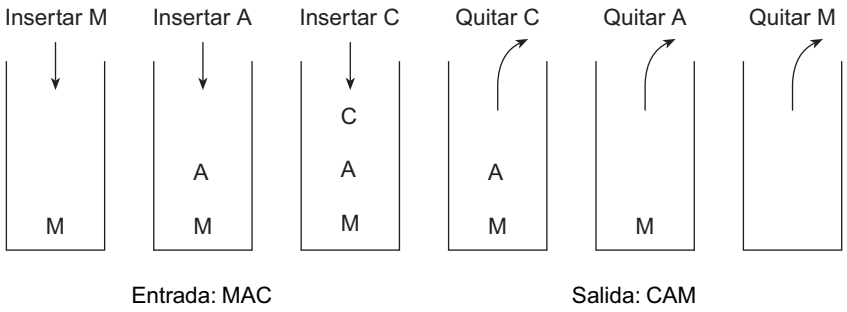


Figura 11.2. Insertar y quitar elementos de la pila.

La operación `insertar` (*push*) sitúa un elemento en la cima de la pila y `quitar` (*pop*) elimina o extrae el elemento cima de la pila.

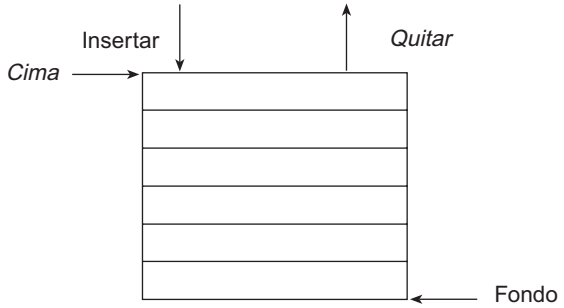


Figura 11.3. Operaciones básicas de una pila.

La pila se puede implementar guardando los elementos en un array, en cuyo caso su dimensión o longitud es fija. Otra forma de implementación consiste en construir una lista enlazada, cada elemento de la pila forma un nodo de la lista; la lista crece o decrece según se añaden o se extraen, respectivamente, elementos de la pila; ésta es una representación dinámica y no existe limitación en su tamaño excepto la memoria del ordenador.

Una pila puede estar *vacía* o *llena* (en la representación con un array, si se ha llegado al último elemento). Si un programa intenta sacar un elemento de una pila vacía, se producirá un error, una *excepción*, debido a que esa operación es imposible; esta situación se denomina **desbordamiento negativo** (*underflow*). Por el contrario, si un programa intenta poner un elemento en una pila *llena* se produce un error, una *excepción*, de **desbordamiento** (*overflow*) o *rebosamiento*. Para evitar estas situaciones se diseñan funciones, que comprueban si la pila está llena o vacía.

11.1.1. Especificaciones de una pila

Las operaciones que sirven para definir una pila y poder manipular su contenido son las siguientes.

Tipo de dato	Dato que se almacena en la pila
Operaciones	
CrearPila	Inicia la pila.
Insertar (push)	Pone un dato en la pila.
Quitar (pop)	Retira (saca) un dato de la pila.
Pilavacía	Comprobar si la pila no tiene elementos.
Pilallena	Comprobar si la pila está llena de elementos.
Limpiar pila	Quita todos sus elementos y dejar la pila vacía.
CimaPila	Obtiene el elemento cima de la pila.
Tamaño de la pila	Número de elementos máximo que puede contener la pila.

La operación *Pilallena* sólo se implementa cuando se utiliza un array para almacenar los elementos. Una pila puede crecer indefinidamente si se implementa con una estructura dinámica.

11.2. TIPO DE DATO PILA IMPLEMENTADO CON ARRAYS

La implementación con un array es *estática* porque el array es de tamaño fijo. La clase *Pila*, con esta representación, además del array, utiliza la variable `cima` para apuntar (índice) al último elemento colocado en la pila. Es necesario controlar el tamaño de la pila para que no exceda al número de elementos del array, y la condición *Pilallena* será significativa para el diseño.

El método usual de introducir elementos en la pila es definir el *fondo* en la posición `0` del array y sin ningún elemento en su interior, es decir, definir una *pila vacía*; a continuación, se van introduciendo elementos en el array de modo que el primer elemento se introduce en una pila vacía y en la posición `0`, el segundo elemento en la posición `1`, el siguiente en la posición `2` y así sucesivamente. Con estas operaciones el índice que apunta a la cima de la pila va incrementando en `1` cada vez que se añade un nuevo elemento. Los algoritmos de insertar (*push*) y quitar (*pop*) datos de la pila son::

Insertar (*push*)

1. Verificar si la pila no está llena.
2. Incrementar en 1 el apuntador (cima) de la pila.
3. Almacenar el elemento en la posición del apuntador de la pila.

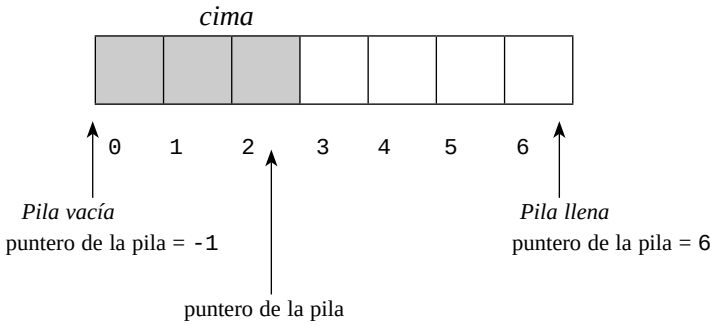
Quitar (*pop*)

1. Si la pila no está vacía.
2. Leer el elemento de la posición del apuntador de la pila.
3. Decrementar en 1 el apuntador de la pila.

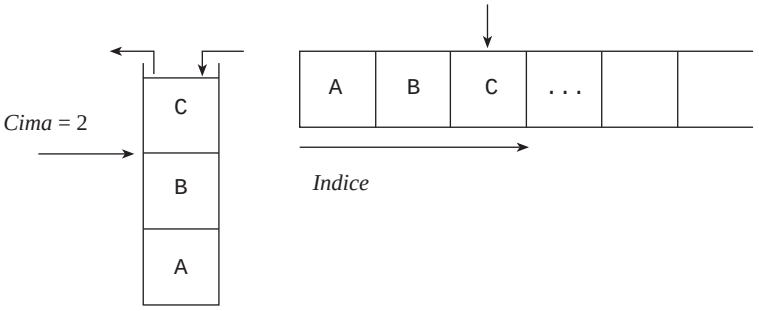
El rango de elementos que puede tener una pila varía de `0` a `TAMPILA-1`, en el supuesto de que el array se defina de tamaño `TAMPILA` elementos. De modo que *en una pila llena* el

apuntador (índice del array) de la pila tiene el valor `TAMPILA-1,` y *en una pila vacía* tendrá `-1` (el valor `0` es el índice del primer elemento).

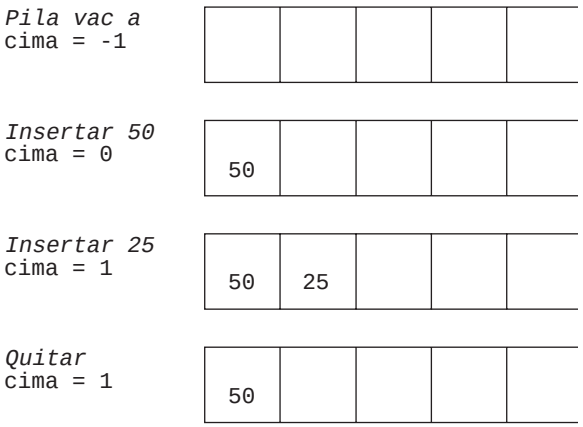
EJEMPLO 11.1. Una pila de 7 elementos se puede representar g ráficamente así:



Si se almacenan los datos `A, B, C`, ... en la pila se puede representar gráficamente de alguna de estas formas



A continuación, se muestra la imagen de una pila según diferentes operaciones realizadas.



11.2.1. Especificación de la clase `Pila`

La declaración de un tipo abstracto incluye la representación de los datos y la definición de las operaciones. En el *TAD* `Pila` los datos pueden ser de cualquier tipo y las operaciones las ya citadas en 11.1.1.

1. Datos de la pila (`TipoDato` es cualquier tipo de dato primitivo o tipo clase).
2. `crearPila` inicializa una pila. Crear una pila sin elementos, por tanto, vacía.
3. Verificar que la pila no está llena antes de `insertar` o poner ("*push*") un elemento en la pila; verificar que una pila no está vacía antes de `quitar` o sacar ("*pop*") un elemento de la pila. Si estas precondiciones no se cumplen se debe visualizar un mensaje de error (una *excepción*) y el programa debe terminar.
4. `pilaVacia` devuelve *verdadero* si la pila está vacía y *falso* en caso contrario.
5. `pilaLlena` devuelve *verdadero* si la pila está llena y *falso* en caso contrario. Estas dos últimas operaciones se utilizan para verificar las precondiciones de *insertar* y *quitar*.
6. `limpiarPila` vacía la pila, dejándola sin elementos y disponible para otras tareas.
7. `cimaPila`, devuelve el valor situado en la cima de la pila, pero no se decrementa el puntero de la pila, ya que la pila queda intacta.

Declaración de la clase `Pila`

```
typedef tipo TipoDeDato;  // tipo de los elementos de la pila
// archivo de cabecera pilalineal.h
const int TAMPILA = 49;
class PilaLineal
{
  private:
    int cima;
    TipoDeDato listaPila[TAMPILA];
  public:
    PilaLineal()
    {
      cima = -1;  // condición de pila vacía
    }
    // operaciones que modifican la pila
    void insertar(TipoDeDato elemento);
    TipoDeDato quitar();
    void limpiarPila();
    // operación de acceso a la pila
    TipoDeDato cimaPila();
    // verificación estado de la pila
    bool pilaVacia();
    bool pilaLlena();
};
```

La declaración realizada está ligada al tipo de los elementos de la pila. Para alcanzar la máxima abstracción, se declara la clase genérica *PilaLineal* de tal forma que el tipo de dato de los elementos se especifica al crear el objeto pila.

EJEMPLO 11.2. Escribir un programa que cree una pila de enteros. Se realicen operaciones de añadir datos a la pila, quitar ...

Se supone implementada la clase pila con el tipo primitivo `int`. El programa crea una pila de números enteros, inserta en la pila elementos leídos del teclado (hasta leer la clave `-1`), a continuación, extrae los elementos de la pila hasta que se vacía. En pantalla deben escribirse los números leídos en orden inverso por la naturaleza de la pila. El bloque de sentencias se encierra en un bloque `try` para tratar errores de desbordamiento de la pila.

```
#include <iostream>
using namespace std;
typedef int TipoDeDato;
#include "pilalineal.h"

int main()
{
  PilaLineal pila;      // crea pila vacía
  TipoDeDato x;
  const TipoDeDato CLAVE = -1;

  cout << "Teclea elemento de la pila(termina con -1)" << endl;
  try {
    do {
      cin >> x;
      pila.insertar(x);
    }while (x != CLAVE);

    // proceso de la pila
    cout << "Elementos de la Pila: " ;
    while (!pila.pilaVacia())
    {
      x = pila.quitar();
      cout << x << " ";
    }
  }
  catch (const char * error)
  {
    cout << "Excepción: " << error;
  }
  return 0;
}
```

11.2.2. Implementación de las operaciones sobre pilas

Las funciones de la clase `Pila` son sencillas de implementar, teniendo en cuenta la característica principal de esta estructura: *inserciones y borrados se realizan por el mismo extremo, la cima de la pila*.

La operación de `insertar` un elemento en la pila, incrementa el apuntador `cima` y asigna el nuevo elemento a la lista. Cualquier intento de añadir un elemento en una pila llena genera una excepción o error debido al "*Desbordamiento de la pila*".

```
void PilaLineal::insertar(TipoDeDato elemento)
{
```

```
  if (pilaLlena())
  {
    throw "Desbordamiento pila";
  }
  //incrementar puntero cima y copia elemento
  cima++;
  listaPila[cima] = elemento;
}
```

La operación `quitar` elimina un elemento de la pila copiando, en primer lugar, el valor de la cima de la pila en una variable local, `aux`, y a continuación, decrementa el puntero de la pila. `quitar()` devuelve la variable `aux`, es decir, el elemento eliminado por la operación. Si se intenta eliminar o borrar un elemento en una pila vacía se produce error, se lanza una excepción.

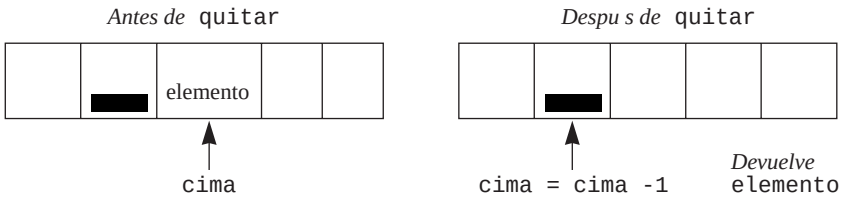


Figura 11. 4. Extraer elemento cima.

```
TipoDeDato PilaLineal::quitar()
{
  TipoDeDato aux;
  if (pilaVacia())
  {
    throw "Pila vacía, no se puede extraer.";
  }
    // guarda elemento de la cima
    aux = listaPila[cima];
    // decrementar cima y devolver elemento
    cima--;
    return aux;
}
```

La operación `cimaPila` devuelve el elemento que se encuentra en la cima de la pila, no se modifica la pila, únicamente accede al elemento.

```
TipoDeDato PilaLineal::cimaPila()
{
  if (pilaVacia())
  {
    throw "Pila vacía, no hay elementos.";
  }
  return listaPila[cima];
}
```

11.2.3. Operaciones de verificación del estado de la pila

Se debe proteger la integridad de la pila, para lo cual el tipo `Pila` ha de proporcionar operaciones que comprueben el estado de la pila: *pila vacía* o *pila llena*. Asimismo, se ha

de definir una operación, `LimpiarPila`, que restaure la condición inicial de la pila, cima igual a -1.

La función `pilaVacia` comprueba si la cima de la pila es -1. En cuyo caso la pila está vacía y devuelve *verdadero*.

```
bool PilaLineal::pilaVacia()
{
  return cima == -1;
}
```

La función `pilaLlena` comprueba si la cima es `TAMPILA-1`; en cuyo caso la pila está llena y devuelve *verdadero*.

```
bool PilaLineal::pilaLlena()
{
  return cima == TAMPILA - 1;
}
```

Por último, `limpiarPila()` pone la cima de la pila a su valor inicial.

```
void PilaLineal::limpiarPila()
{
  cima = -1;
}
```

EJERCICIO 11.1. Escribir un programa que utilice una `Pila` para comprobar si una determinada frase/palabra (cadena de caracteres) es un palíndromo. Nota: una palabra o frase es un palíndromo cuando la lectura directa e indirecta de la misma tiene igual valor: **alila**, es un palíndromo; **cara (arac)** no es un palíndromo.

La palabra se lee con la función `gets()` y se almacena en un `string`; cada carácter de la palabra se pone en una pila de caracteres. Una vez leída la palabra y construida la pila, se compara el primer carácter del `string` con el carácter que se extrae de la pila, si son iguales sigue la comparación con el siguiente carácter del `string` y de la pila; así sucesivamente hasta que la pila se queda vacía o hay un carácter no coincidente.

Al guardar los caracteres de la palabra en la pila se garantiza que las comparaciones de caracteres se realizan en orden inverso: primero con último … .

No es necesario volver a implementar las operaciones de la clases `Pila`, simplemente se cambia el tipo de dato de los elementos, en esta ocasión `char`.

```
#include <iostream>
using namespace std;
#include <string.h>
typedef char TipoDeDato;
#include "pilalineal.h"

int main()
{
  PilaLineal pilaChar;   // crea pila vacía
  TipoDeDato ch;
  bool esPal;
  char pal[81];
```

```
  cout << "Teclea la palabra verificar si es palíndromo: ";
  gets(pal);
  for (int i = 0; i < strlen(pal); )
  {
    char c;
    c = pal[i++];
    pilaChar.insertar(c);
  }
  // se comprueba si es palíndromo

  esPal = true;
  for (int j = 0; esPal && !pilaChar.pilaVacia(); )
  {
    char c;
    c = pilaChar.quitar();
    esPal = pal[j++] == c;
  }
  pilaChar.limpiarPila();
  if (esPal)
    cout << "La palabra " << pal << " es un palíndromo \n";
  else
    cout << "La palabra " << pal << " no es un palíndromo \n";
  return 0;
}
```

EJEMPLO 11.3. Llenar una pila con números leídos del teclado. A continuación vaciar la pila de tal forma que se muestren los valores positivos.

En este ejemplo el tipo de los elementos de la pila es `double`. El número de elementos que tendrá la pila se solicita al usuario; en un bucle *for* se lee el elemento y se inserta en la pila. Para vaciar la pila se diseña un bucle, *hasta pila vacía*; cada elemento que se extrae se escribe si es positivo.

```
#include <iostream>
using namespace std;
typedef double TipoDeDato;
#include "pilalineal.h"

int main()
{
  PilaLineal pila;
  int x;

  cout << "Teclea número de elementos: ";
  cin >> x;
  for (int j = 1; j <= x; j++)
  {
    double d;
    cin >> d;
    pila.insertar(d);
  }
  // vaciado de la pila
```

```
  cout << "Elementos de la Pila: ";
  while (!pila.pilaVacia())
  {
    double d;
    d = pila.quitar();
    if (d > 0.0)
      cout << d << " ";
  }
  return 0;
}
```

11.3. PILA GENÉRICA CON LISTAS ENLAZADA

La realización dinámica de una pila utilizando una lista enlazada almacena cada elemento de la pila como un nodo de la lista. Como las operaciones de *insertar* y *extraer* en el *TAD Pila* se realizan por el mismo extremo (cima de la pila), las acciones correspondientes con la lista se realizarán siempre por el mismo extremo de la lista.

Esta realización tiene la ventaja de que el tamaño se ajusta exactamente al número de elementos de la pila. Sin embargo, para cada elemento es necesaria más memoria para guardar el campo de enlace entre nodos consecutivos. La Figura 11.5 muestra la imagen de una pila implementada con una lista enlazada.

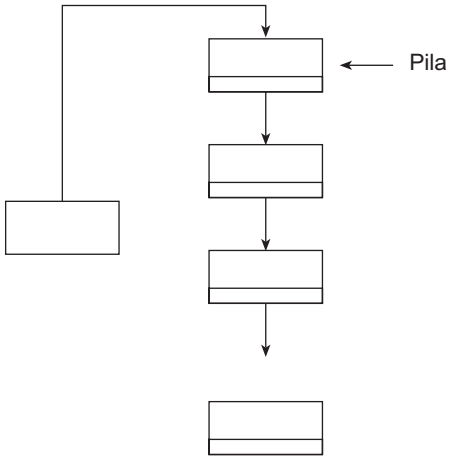


Figura 11.5. Representación de una pila con una lista enlazada.

Nota

Una pila realizada con una lista enlazada crece y decrece dinámicamente . En tiempo de ejecución, se reserva memoria según se ponen elementos en la pila y se libera memoria según se extraen elementos de la pila.

11.3.1. Clase *PilaGenerica* y *NodoPila*

La estructura que tiene la pila implementada con una lista enlaza es muy similar a la expuesta en listas enlazadas. Los elementos de la pila son los nodos de la lista, con un atributo para guardar el elemento y otro de enlace. Las operaciones del tipo pila implementada con listas son, naturalmente, las mismas que si la pila se implementa con arrays, salvo la operación que controla si la pila está llena, *pilaLlena*, que ahora no tiene significado ya que la listas enlazadas crecen indefinidamente, con el único límite de la memoria.

El tipo de dato de elemento se corresponde con el tipo de los elementos de la `Pila`, para que no dependa de un tipo concreto; para que sea genérico, se diseña una *pila genérica* utilizando las *plantillas* (`template`) de C++. La clase `NodoPila` representa un nodo de la lista enlazada, tiene dos atributos: elemento, guarda el elemento de la pila y siguiente, contiene la dirección del siguiente nodo de la lista. En esta implementación, `NodoPila` es una clase interna de `PilaGenerica`.

```
// archivo PilaGenerica.h

template <class T>
class PilaGenerica
{
  class NodoPila
  {

    public:
      NodoPila* siguiente;
      T elemento;
      NodoPila(T x)
      {
        elemento = x;
        siguiente = NULL;
      }
  };
  NodoPila* cima;

  public:
    PilaGenerica ()
    {
      cima = NULL;
    }
    void insertar(T elemento);
    T quitar();
    T cimaPila(); const
    bool pilaVacia(); const
    void limpiarPila();
    ~PilaGenerica()
    {
      limpiarPila();
    }
};
```

11.3.2. Implementación de las operaciones del *TAD Pila* con listas enlazadas

El constructor de `Pila` inicializa a ésta como pila vacía `(cima == NULL)`, realmente, a la condición de *lista vacía*. Las operaciones `insertar`, `quitar` y `cimaPila` acceden a la lista directamente con el puntero cima (apunta al último nodo apilado). Entonces, como no necesitan recorren los nodos de la lista, no dependen del número de nodos, la eficiencia de cada operación es constante, `O(1)`.

La codificación que a continuación se escribe, es para una pila de elemento de cualquier tipo. Es preciso recordar que al crear una instancia de pila es cuando se informa del tipo concreto de sus elementos, por ejemplo `PilaGenerica<char> pila`.

Verificación del estado de la pila

```
template <class T>
bool PilaGenerica<T>::pilaVacia() const
{
   return cima == NULL;
}
```

Poner un elemento en la pila

Crea un nuevo nodo con el elemento que se pone en la pila y se enlaza por la cima.

```
template <class T>
void PilaGenerica<T>::insertar(T elemento)
{
  NodoPila* nuevo;
  nuevo = new NodoPila(elemento);
  nuevo -> siguiente = cima;
  cima = nuevo;
}
```

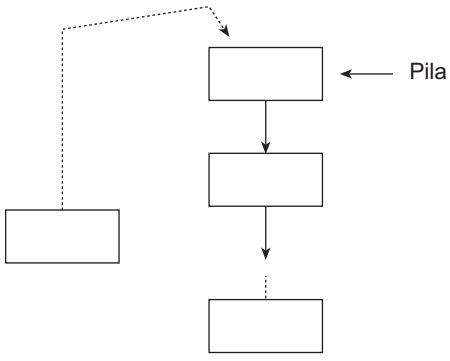


Figura 11.6. Apilar un elemento.

Eliminación del elemento cima

Retorna el elemento cima y lo quita de la pila, disminuye el tamaño de la pila.

```
template <class T>
T PilaGenerica<T>::quitar()
{
  if (pilaVacia())
    throw "Pila vacía, no se puede extraer.";
  T aux = cima -> elemento;
  cima = cima -> siguiente;
  return aux;
}
```

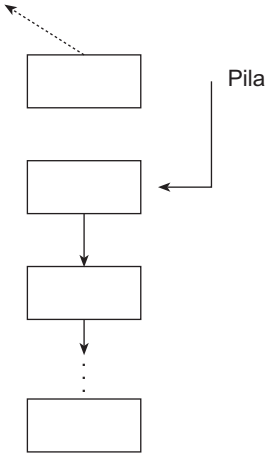


Figura 11.7. Quita la cima de la pila.

Obtención del elemento cima de la pila

```
template <class T>
T PilaGenerica<T>:: cimaPila(); const
{
  if (pilaVacia())
    throw "Pila vacía";
  return cima -> elemento;
}
```

Vaciado de la pila

Libera todos los nodos de que consta la pila. Recorre los `n` nodos de la lista enlazada, es una operación de complejidad lineal, `O(n)`.

```
template <class T>
void PilaGenerica<T>:: limpiarPila()
```

```
{
  NodoPila* n;
  while(!pilaVacia())
  {
    n = cima;
    cima = cima -> siguiente;
    delete n;
  }
}
```

11.4. EVALUACIÓN DE EXPRESIONES ARITMÉTICAS CON PILAS

Una *expresión aritmética* está formada por operandos y operadores. La expresión `x * y - (a + b)` consta de los operadores `*, -, +` y de los operandos `x, y, a, b`. Los operandos pueden ser valores constantes, variables o, incluso, otra expresión. Los operadores son los símbolos conocidos de las operaciones matemáticas.

La evaluación de una expresión aritmética da lugar a un valor numérico, se realiza sustituyendo los operandos que son variables por valores concretos y ejecutando las operaciones aritméticas representadas por los operadores. Si los operandos de la expresión anterior toman los valores: `x = 5, y = 2, a = 3, b = 4` el resultado de la evaluación es:

```
5 * 2 – (3 + 4) = 5 * 2 – 7 = 10 – 7 = 3
```

La forma habitual de escribir expresiones matemáticas sitúa el operador entre sus dos operandos. La expresión anterior está escrita de esa forma, recibe el nombre de *notación infija*. Esta forma de escribir las expresiones exige, en algunas ocasiones, el uso de paréntesis para *encerrar* subexpresiones con mayor prioridad.

Los operadores, como es sabido, tienen distintos niveles de precedencia o prioridad a la hora de su evaluación. A continuación, se recuerda estos niveles de prioridad en orden de mayor a menor:

```
Paréntesis              : ( )
Potencia                : ^
Multiplicación/división : *, /
Suma/Resta              : +, -
```

Normalmente, en una expresión hay operadores con la misma prioridad, a igualdad de precedencia, los operadores se evalúan de izquierda a derecha *(asociatividad)*, excepto la potencia que es de derecha a izquierda.

11.4.1. Notación *prefija* y notación *postfija* de una expresiones aritmética

Las operaciones aritméticas escritas en *notación infija* en muchas ocasiones necesitan usar paréntesis para indicar el orden de evaluación. Las expresiones

```
r = a * b / (a + c)                    g = a * b / a + c
```

son distintas al no poner paréntesis en la expresión `g`. Igual ocurre con estas otras:

```
r = (a - b) ^ c + d                    g = a - b ^ c + d
```

Existen otras formas de escribir expresiones aritméticas, que se diferencian por la ubicación del operador respecto de los operandos. La notación en la que el operador se coloca delante de los dos operandos, *notación prefija*, se conoce también como *notación polaca* por el matemático polaco que la propuso. En el Ejemplo 11.4 se escriben expresiones en notación *prefija* o notación polaca.

EJEMPLO 11.4. Dadas las expresiones: `a * b / (a + c); a * b / a + c; (a-b)^c+d`. Escribir las expresiones equivalentes en notación prefija.

Paso a paso, se escribe la transformación de cada expresión algebraica en la expresión equivalente en notación polaca.

```
a * b / (a + c) (infija) → a * b / + ac → * ab / + ac → / *ab + ac (polaca)
a * b / a + c (infija) → * ab / a + c → / * aba + c → + / *abac (polaca)
(a - b) ^ c + d (infija) → -ab ^ c + d → ^ -abc + d → +^-abcd (polaca)
```

Nota

La propiedad fundamental de la notación polaca es que el orden de ejecución de las operaciones está determinado por las posiciones de los operadores y los operandos en la expresión. No son necesarios los paréntesis al escribir la expresión en notación polaca, como se observa en el Ejemplo 11.4.

Notación postfija

Hay más formas de escribir las expresiones. La notación *postfija* o *polaca inversa* coloca el operador a continuación de sus dos operandos.

EJEMPLO 11.5. Dadas las expresiones: `a*b/(a+c); a*b/a+c; (a-b)^c+d`. Escribir las expresiones equivalentes en notación postfija.

Paso a paso se transforma cada subexpresión en notación polaca inversa.

```
a*b/(a+c) (infija) → a*b/ac+ → ab*/ac+ → ab*ac+/    (polaca inversa)
a*b/a+c (infija) → ab*/a+c → ab*a/+c → ab*a/c+      (polaca inversa)
(a-b)^c+d (infija) → ab-^c+d → ab-c^+d → ab-c^d+    (polaca inversa)
```

Recordar

Las diferentes formas de escribir una misma expresión algebraica dependen de la ubicación de los operadores respecto a los operandos. Es importante tener en cuenta que tanto en la notación prefija como en la postfija no son necesarios los paréntesis para cambiar el orden de evaluación.

11.4.2. Evaluación de una expresión aritmética

La evaluación de una expresión aritmética escrita de *manera habitual*, en *notación infija*, se realiza en dos pasos principales:

1.º Transformar la expresión de notación infija a postfija.
2.º Evaluar la expresión en notación postfija.

El *TAD Pila* es fundamental en los algoritmos que se aplican a cada uno de los pasos. El orden que fija la estructura pila asegura que el *último en entrar es el primero en salir*, y de esa forma el algoritmo de transformación a *postfija* sitúa los operadores después de sus operandos, con la prioridad o precedencia que le corresponde. Una vez que se tiene la expresión en notación *postfija*, se utiliza otra pila, de elementos numéricos, para guardar los valores de los operandos, y de las operaciones parciales con el fin de obtener el valor numérico de la expresión.

11.4.3. Transformación de una expresión infija a *postfija*

Se parte de una expresión en *notación infija* que tiene operandos, operadores y puede tener paréntesis. Los operandos se representan con letras, los operadores son éstos:

```
^ (potenciación), *, /, +, - .
```

La transformación se realiza utilizando una pila para guardar operadores y los paréntesis izquierdos. La expresión aritmética se lee del teclado y se procesa carácter a carácter. Los operandos pasan directamente a formar parte de la expresión en *postfija* la cual se guarda en un array. Un operador se mete en la pila si se cumple que:

- La pila esta vacía, o,
- El operador tiene mayor prioridad que el operador cima de la pila, o bien,
- El operador tiene igual prioridad que el operador cima de la pila y se trata de la máxima prioridad.

Si la prioridad es menor o igual que la de *cima pila*, se saca el elemento cima de la pila, se pone en la expresión en *postfija* y se vuelve a hacer la comparación con el nuevo elemento cima.

El paréntesis izquierdo siempre se mete en la pila; ya en la pila se les considera de mínima prioridad para que todo operador que se encuentra dentro del paréntesis entre en la pila. Cuando se lee un paréntesis derecho se sacan todos los operadores de la pila y pasan a la expresión *postfija*, hasta llegar a un paréntesis izquierdo que se elimina ya que los paréntesis no forman parte de la expresión *postfija*. El algoritmo termina cuando no hay más items de la expresión origen y la pila está vacía.

Por ejemplo, dada la expresión `a*(b+c-(d/e^f)-g)-h` escrita en *notación infija*, a continuación, se va a ir formando, paso a paso, la expresión equivalente en postfija.

Expresión en postfija

`a`	Operando `a` pasa a la expresión *postfija*; operador `*` a la pila.
`ab`	Operador `(` pasa a la pila; operando `b` a la expresión.

`abc` Operador + pasa a la pila; operando a la expresión.

En este punto, el estado de la pila:

El siguiente carácter de la expresión, `-`, tiene igual prioridad que el operador de la cima (+), da lugar:

`abc+`

`abc+d` El operador `(`se mete en la pila; el operando `d` a la expresión.

`abc+de` El operador `/` pasa a la pila; el operando e a la expresión.

`abc+def` El operador `^` pasa a la pila; el operando `f` a la expresión.
El siguiente item,`)` (paréntesis derecho), produce que se vacié la pila hasta un `(` .La pila, en este momento, dispone de estos operadores:

`abc+def^/` El algoritmo saca operadores de la pila hasta un '(' y da lugar a la pila:

`abc+def^/-` El operador – pasa a la pila y, a su vez, se extrae `-`; el siguiente carácter, el operando `g` pasa a la expresión.

`abc+def^/-g` El siguiente carácter es `)`, por lo que son extraídos de la pila los operadores hasta un `(`, la pila queda de la siguiente forma:

`abc+def^/-g-` El siguiente carácter es el operador `-`, hace que se saque de la pila el operador * y se meta en la pila el operador –.

`abc+def^/-g-*` Por último, el operando h pasa directamente a la expresión.

`abc+def^/-g-*h` Fin de entrada, se vacía la pila pasando los operadores a la expresión:
`abc+def^/-g-*h-`

El seguimiento realizado pone de manifiesto la importancia de considerar al paréntesis izquierdo un operador de mínima prioridad dentro de la pila, para que los operadores, dentro de un paréntesis, se metan en la pila y después extraerlos cuando se trata el paréntesis derecho. También tiene un comportamiento distinto el operador de potenciación dentro y fuera de la pila, debido a que tienen asociatividad de derecha a izquierda. Las prioridades se fijan en la Tabla 11.1.

Tabla 11.1. Tabla de prioridades de los operadores considerados.

Operador	Prioridad dentro pila	Prioridad fuera pila
^	3	4
*, /	2	2
+, -	1	1
(	0	5

Observe, que el paréntesis derecho no se considera ya que éste provoca sacar operadores de la pila hasta el paréntesis izquierdo.

Algoritmo de paso de notación *infija* a *postfija*

Los pasos a seguir para transformar una expresión algebraica de *notación infija* a *postfija*:

1. Obtener caracteres de la expresión y repetir los pasos 2 al 4 para cada carácter.
2. Si es un operando, pasarlo a la expresión postfija.
3. Si es operador:
   3.1. Si la pila está vacía, meterlo en la pila. Repetir a partir de 1.
   3.2. Si la pila no está vacía:
   Si prioridad del operador es mayor que prioridad del operador cima, meterlo en la pila y repetir a partir de 1.
   Si prioridad del operador es menor o igual que prioridad del operador cima, sacar operador cima de la pila y ponerlo en la expresión postfija, volver a 3.
4. Si es paréntesis derecho:
   4.1. Sacar operador cima y ponerlo en la expresión postfija.
   4.2. Si el nuevo operador cima es paréntesis izquierdo, suprimir elemento cima.
   4.3. Si cima no es paréntesis izquierdo, volver a 4.1.
   4.4. Volver a partir de 1.
5. Si quedan elementos en la pila pasarlos a la expresión postfija.
6. Fin del algoritmo.

Codificación del algoritmo de transformación a *postfija*

Se necesita crear una pila de caracteres para guardar los operadores. Se utiliza el diseño de Pila genérica del Apartado 11.3. La expresión original se lee del teclado en una cadena de suficiente tamaño. También se declara una estructura para representar un elemento de la ex-

presión, con un campo carácter para el operando o el operador, y el otro campo para indicar si es operador u operando.

```
struct Elemento
{
  char c;
  bool oprdor;
};
```

La función `postFija()` implementa los pasos del algoritmo de transformación, recibe como argumento una cadena con la expesión, crea una pila y en un bucle de tantas iteraciones como caracteres realiza las acciones del algoritmo. La función define el array `Elemento* expresión`, a la que se pasan los elementos que forman la expresión en postfija. Una vez que termina la transformación, la función devuelve la expresión en *notación postfija* y el número de elementos de que consta agrupados en la siguiente estructura:

```
struct Expresion
{
  Elemento* expr;
  int n;
};
```

El archivo `TiposExpresio.h` contiene el tipo `Elemento` y el tipo `Expresion`.

En la función `prdadDentro()`se fija la prioridad de un operador dentro de la pila, y en `prdadFuera()`la prioridad de un operador fuera de la pila.

```
//archivo postFija.cpp

#include <cstdlib>
#include <string.h>
#include <ctype.h>
#include "PilaGenerica.h"
#include "TiposExpresio.h"

Expresion postFija(const char* expOrg);
int prdadFuera (char op);
int prdadDentro (char op);
bool valido(const char* expresion);
bool operando(char c);

Expresion postFija(const char* expOrg)
{
  PilaGenerica<char> pila;
  Elemento* expsion;
  bool desapila;
  int n = -1;   //contador de expresión en postfija

  if (! valido(expOrg))  // verifica los caracteres de la expresión
     throw "Carácter no válido en una expresión";

  expsion = new Elemento[strlen(expOrg)];
  for (int i = 0; i < strlen(expOrg); i++)
  {
    char ch, opeCima;
    ch = toupper(expOrg[i]);    // operandos en mayúsculas
```

```
    if (operando(ch))           // análisis del elemento
    {
      expsion[++n].c = ch;
      expsion[n].oprdor = false;
    }
    else if (ch != ')')         // es un operador
    {
      desapila = true;
      while (desapila)
      {
        opeCima = ' ';
        if (!pila.pilaVacia())
          opeCima = pila.cimaPila();
          if (pila.pilaVacia() ||
            (prdadFuera(ch) > prdadDentro(opeCima)))
          {
            pila.insertar(ch);
            desapila = false;
          }
          else if (prdadFuera(ch) <= prdadDentro(opeCima))
          {
            expsion[++n].c = pila.quitar();
            expsion[n].oprdor = true;
          }
        }
    }
    else                        // es un ')'
    {
      opeCima = pila.quitar();
      do{
        expsion[++n].c = opeCima;
        expsion[n].oprdor = true;
        opeCima = pila.quitar();
      }while (opeCima != '(');
    }
  }
    /*
       se vuelca los operadores que quedan en la pila y se pasan a la ex-
       presión.
    */
  while (!pila.pilaVacia())
  {
    expsion[++n].c = pila.quitar();
    expsion[n].oprdor = true;
  }
  Expresion post;
  post.expr = expsion;
  post.n = n;
  return post;   // expresión en postfija
}
// prioridad del operador dentro de la pila

int prdadDentro(char op)
{
  int pdp;
  switch (op)
```

```
  {
    case '^':  pdp = 3;
               break;
    case '*': case '/':
               pdp = 2;
               break;
    case '+': case '-':
               pdp = 1;
               break;
    case '(':  pdp = 0;
  }
  return pdp;
}
// prioridad del operador en la expresión infija
int prdadFuera(char op)
{
  int pfp;
  switch (op)
  {
    case '^':  pfp = 4;
               break;
    case '*': case '/':
               pfp = 2;
               break;
    case '+': case '-':
               pfp = 1;
               break;
    case '(':  pfp = 5;
  }
  return pfp;
}

//analiza cada carácter de la expresión
bool valido(const char* expresion)
{
  bool sw = true;
  for (int i = 0; (i < strlen(expresion))&& sw; i++)
  {
    char c;
    c = expresion[i];
    sw = sw && (
                (c >= 'A' && c <= 'Z') ||
                (c >= 'a' && c <= 'z') ||
                (c == '^' || c == '/' || c== '*' ||
                  c == '+' || c == '-' || c== '\n' ||
                  c == '(' || c == ')' )
               );
  }
  return sw;
}

bool operando(char c)
{

//determina si c es un operando
  return (c >= 'A' && c <= 'Z');
}
```

11.4.4. Evaluación de la expresión en notación *postfija*

Una vez que se tiene la expresión en *notación postfija* se evalúa la expresión. Evaluar significa obtener un resultado de la expresión para valores particulares de los operandos. De nuevo, el algoritmo de evaluación utiliza una pila, en esta ocasión de operandos, es decir, de números reales.

Al describir el algoritmo `expsion` es el array con la la expresión postfija. El número de elementos es la longitud, `n`, de la cadena.

1. Examinar `expsion` elemento a elemento: repetir los pasos 2 y 3 para cada elemento.
2. Si el elemento es un operando, meterlo en la pila.
3. Si el elemento es un operador, se simboliza con `&`, entonces:
   - Sacar los dos elementos superiores de la pila, se denominarán `b` y `a` respectivamente.
   - Evaluar `a & b`, el resultado es `z = a & b`.
   - El resultado `z`, meterlo en la pila. Repetir a partir del paso 1.
4. El resultado de la evaluación de la expresión está en el elemento cima de la pila.
5. Fin del algoritmo.

Codificación de la evaluación de expresión en postfija

La función `double evalua()` implementa el algoritmo, recibe la expresión en postfija y el array con el valor de cada operando. La pila de números reales, utilizada por el algoritmo, se instancia de la clase genérica `PilaGenerica` para elementos de tipo `double`.

La función `valorOprdos()` da entrada a los valores de los operandos. Estos valores se guardan en un array, cada posición del array se corresponde con una letra, que a su vez es un operando.

```
// archivo evaluaExpresion.h

#include "pilagenerica.h"
#include "TiposExpresio.h"
#include <math.h>

double evalua(Expresion postFija, double v[]);
void valorOprdos(Expresion ep, double v[]);

double evalua(Expresion postFija, double v[])
{
  PilaGenerica<double> pila;
  double valor, a, b;

  for (int i = 0; i <= postFija.n; i++)
  {
    char op;

    if (postFija.expr[i].oprdor)     // es un operador
    {
      op = postFija.expr[i].c;
```

```
        b = pila.quitar();
        a = pila.quitar();
        switch (op)
        {
          case '^': valor = pow(a,b);
                    break;
          case '*': valor = a * b;
                    break;
          case '/': if (b != 0.0)
                      valor = a / b;
                    else
                      throw "División por cero.";
                    break;
          case '+': valor = a + b;
                    break;
          case '-': valor = a – b;
        }
        pila.insertar(valor);
      }
      else               // es un operando
      {
        int indice;
        op = postFija.expr[i].c;
        indice = op - 'A';       // posición en array de valores
        pila.insertar(v[indice]);
      }
    }
    return pila.quitar();        // resultado de la expresión
  }

  // asignan valores numéricos a los operandos
  void valorOprdos(Expresion ep, double v[])
  {
    char ch;
    for (int i = 0; i <= ep.n; i++)
    {
      char op;
      op = ep.expr[i].c;
      if (! ep.expr[i].oprdor)   // es un operando
      {
        int indice;
        double d;
        indice = op - 'A';
        cout << op << " = ";
        cin >> v[indice];
      }
    }
  }
```

Programa

La función `main()` controla las etapas principales del algoritmo: petición de la expresión algebraica, llamada a la función que transforma a notación postfija y, por último, evaluar la expresión para unos valores concretos de los operandos.

```
#include <iostream>
using namespace std;
#include "TiposExpresio.h"

Expresion postFija(const char* expOrg);
double evalua(Expresion postFija, double v[]);
void valorOprdos(Expresion ep, double v[]);

int main()
{
  double v[26];
  double resultado;
  char expresion[81];
  Expresion ex;

  cout << "\nExpresión aritmética: ";
  cin.getline(expresion, 80);
  // Conversión de infija a postfija
  expr = postFija(expresion);
  cout << "\nExpresión en postfija: ";
  for (int i = 0; i <= ex.n; i++)
  {
    cout << ex.expr[i];
  }
  // Evaluación de la expresión
  valorOprdos(ex, v);       // valor de operandos
  resultado = evalua(ex, v);
  cout << "Resultado = " << resultado;
  return 0;
}
```

RESUMEN

Una *pila* es una estructura de datos tipo **LIFO** (*last in first out*, último en entrar primero en salir) en la que los datos (todos del mismo tipo) se añaden y eliminan por el mismo extremo, denominado *cima* de la pila. Se definen las siguientes operaciones básicas sobre pilas: `crear`, `insertar`, `cima`, `quitar`, `pilaVacia`, `pilaLlena` y `limpiarPila`.

`insertar`, añade un elemento en la cima de la pila. Debe de haber espacio en la pila.
`cima`, devuelve el elemento que está en la cima, sin extraerlo.
`quitar`, extrae de la pila el elemento cima de la pila.
`pilaVacia`, determina si el estado de la pila es vacía.
`pilaLlena`, determina si existe espacio en la pila para añadir un nuevo elemento.
`limpiarPila`, el espacio asignado a la pila se libera, queda disponible.

Las aplicaciones de las pilas en la programación son numerosas, entre las que está la evaluación de expresiones aritméticas. Primero, se transforma la expresión a notación postfija, a continuación se evalúa.

Las expresiones en notación polaca, postfija o prefija, tienen la característica de que no necesitan paréntesis.

EJERCICIOS

11.1. ¿ Cuál es la salida de este segmento de código, teniendo en cuenta que el tipo de dato de la pila es `int`:

```
Pila p;
int x = 4, y;
p.insertar(x);
cout << "\n " << p.cimaPila();
y = p.quitar();
p.insertar(32);
p.insertar(p.quitar());
do {
   cout << "\n " << p.quitar();
}while (!p.pilaVacia());
```

11.2. Con las operaciones implementadas en la clase `PilaGenerica` escribir la función `mostrarPila()` que muestre en pantalla los elementos de una pila de cadenas.

11.3. Utilizando una pila de caracteres, transformar la siguiente expresión a su equivalente expresión en postfija.

```
(x-y)/(z+w) – (z+y)^x
```

11.4. Obtener una secuencia de 10 números reales, guardarlos en un array y ponerlos en una pila. Imprimir la secuencia original y, a continuación, imprimir la pila extrayendo los elementos.

11.5. Transformar la expresión algebraica del Ejercicio 11.3 en su equivalente expresión en *notación prefija*.

11.6. Dada la expresión algebraica `r = x * y - (z + w)/(z + y) ^ x`, transformar la expresión a *notación postfija* y, a continuación, evaluar la expresión para los valores: `x = 2, y = -1, z = 3, w = 1`. Obtener el resultado de la evaluación siguiendo los pasos del algoritmo descrito en el Apartado 11.4.3.

11.7. Se tiene una lista enlazada a la cual se accede por el primer nodo. Escribir una función que visualice los nodos de la lista en orden inverso, desde el último nodo al primero; como estructura auxiliar utilizar una pila y sus operaciones.

11.8. La implementación del *TAD* `Pila` con arrays establece un tamaño máximo de la pila que se controla con la función `pilaLlena()`. Modificar la función de tal forma que cuando se llene la pila se amplíe el tamaño del array a justamente el doble de la capacidad actual.

PROBLEMAS

11.1. Escribir una función, `copiarPila()`, que copie el contenido de una pila en otra. La función tendrá dos argumentos de tipo pila, uno la pila fuente y otro la pila destino. Utilizar las operaciones definidas sobre el *TAD* `Pila`.

11.2. Escribir una función para determinar si una secuencia de caracteres de entrada es de la forma:

```
X & Y
```

donde `X` una cadena de caracteres e `Y` la cadena inversa. El carácter `&` es el separador.

11.3. Escribir un programa que haciendo uso de una `Pila`, procese cada uno de los caracteres de una expresión que viene dada en una línea. La finalidad es verificar el equilibrio de paréntesis, llaves y corchetes. Por ejemplo, la siguiente expresión tiene un número de paréntesis equilibrado:

```
((a+b)*5) - 7
```

y esta otra expresión le falta un corchete: `2*[(a+b)/2.5+x -7*y`

11.4. Escribir un programa en el que se manejen un total de n = 5 pilas: P_1, P_2, P_3, P_4 y P_5. La entrada de datos será pares de enteros `(i,j)` tal que `1 ≤ abs(i) ≤ n`. De tal forma que el criterio de selección de pila:

- Si i es positivo, debe de insertarse el elemento j en la pila P_i .
- Si i es negativo, debe de eliminarse el elemento j de la pila P_i .
- Si i es cero, fin del proceso de entrada.

Los datos de entrada se introducen por teclado. Cuando termina el proceso el programa debe escribir el contenido de las n pilas en pantalla.

11.5. Modificar el Programa 11.4 para que la entrada sean triplas de números enteros `(i,j,k)`, donde `i`, `j` tienen el mismo significado que en 11.4, y `k` es un número entero que puede tomar los valores `–1`, `0` con este significado:

- `-1`, hay que borrar todos los elementos de la pila.
- `0`, el proceso es el indicado en 11.4 con `i` y `j`.

11.6. Se quiere determinar frases que son palíndromo. Para lo cual se ha de seguir la siguiente estrategia: considerar cada línea de una frase; añadir cada carácter de la frase a una pila y a la vez a lista enlazada circular por el final; extraer carácter a carácter, simultáneamente de la pila y de la lista circular el considerado *primero* y su comparación determina si es palíndromo o no. Escribir un programa que lea líneas de y determine si son palíndromo.

11.7. La función de Ackerman, definida de la siguiente forma:

A(m, n) = n + 1	*si m = 0*
A(m, n) = A (m – 1, 1)	*si n = 0*
A(m, n) = A(m – 1, A(m, n – 1))	*si m > 0, y n > 0*

Se obseva que la definición es recursiva y, por consiguiente, la implementación recursiva es inmediata de codificar. Como alternativa, escribir una función que evalúe la función Ackermannç iterativamente utilizando el *TAD Pila*.

Capítulo 12
Colas

Objetivos

Con el estudio de este capítulo usted podrá:

- Especificar el tipo abstracto de datos *Cola*.
- Encontrar las diferencias fundamentales entre *Pilas* y *Colas.*
- Definir una clase *Cola* con arrays.
- Definir una clase *Cola* con listas enlazadas.
- Aplicar el tipo abstracto *Cola* para la resolución de problemas.

Contenido

Conceptos clave

- Array *circular*.
- Cola de objetos.
- Lista enlazada.
- Lista FIFO.
- Prioridad.

INTRODUCCIÓN

En este capítulo se estudia el tipo abstracto de datos *Cola*, estructura muy utilizada en la vida cotidiana, y también para resolver problemas en programación. Esta estructura, al igual que las pilas, almacena y recupera sus elementos atendiendo a un estricto orden. Las colas se conocen como estructuras **FIFO** (*First-in*, *First-out*, primero en entrar - primero en salir), debido a la forma y orden de inserción y de extracción de elementos de la cola. Las colas tienen numerosas aplicaciones en el mundo de la computación: colas de mensajes, colas de tareas a realizar por una impresora, colas de prioridades.

12.1. CONCEPTO DE COLA

Una **cola** es una estructura de datos que almacena elementos en una lista y el acceso a los datos se hace por uno de los dos extremos de la lista. (Figura 12.1). Un elemento se inserta en la cola (parte *final*) de la lista y se suprime o elimina por el frente (parte inicial, *frente*) de la lista. Las aplicaciones utilizan una cola para almacenar elementos en su orden de aparición o concurrencia.

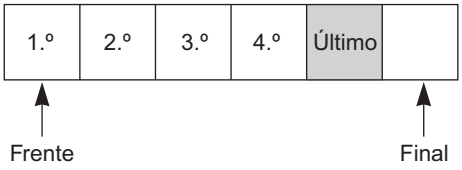


Figura 12.1. Una cola.

Los elementos se eliminan (se quitan) de la cola en el mismo orden en que se almacenan y, por consiguiente, una cola es una estructura de tipo **FIFO** *(first-in/firs-out, primero en entrar/primero en salir* o bien *primero en llegar/primero en ser servido).* El servicio de atención a clientes en un almacén es un típico ejemplo de cola. La acción de gestión de memoria intermedia *(buffering)* de trabajos o tareas de impresora en un distribuidor de impresoras *(spooler)* es otro ejemplo de cola[1]. Dado que la impresión es una tarea (un trabajo) que requiere más tiempo que el proceso de la transmisión real de los datos desde la computadora a la impresora, se organiza una cola de trabajos de modo que los trabajos se imprimen en el mismo orden en que se recibieron por la impresora. Este sistema tiene el gran inconveniente de que si su trabajo personal consta de una única página para imprimir y delante de su petición de impresión existe otra petición para imprimir un informe de 300 páginas, deberá esperar a la impresión de esas 300 páginas antes de que se imprima su página.

Definición

Una cola es una estr uctura de datos cuy os elementos mantienen un cier to orden, tal que sólo se pueden añadir elementos por un extremo, **final** de la cola, y eliminar o extraer por el otro extremo, llamado **frente**.

[1] Recordemos que este caso sucede en sistemas multiusuario donde hay varios terminales y sólo una impresora de servicio. Los trabajos se "encolan" en la cola de impresión.

Las operaciones usuales en las colas son `Insertar` y `Quitar`. La operación `Insertar` añade un elemento por el extremo `final` de la cola, y la operación `Quitar` elimina o extrae un elemento por el extremo opuesto, el frente o primero de la cola. La organización de elementos en forma de cola asegura que *el primero en entrar es el primero en salir*. En la Figura 12.2 se realizan las operaciones básicas sobre colas, insertar y retirar elementos.

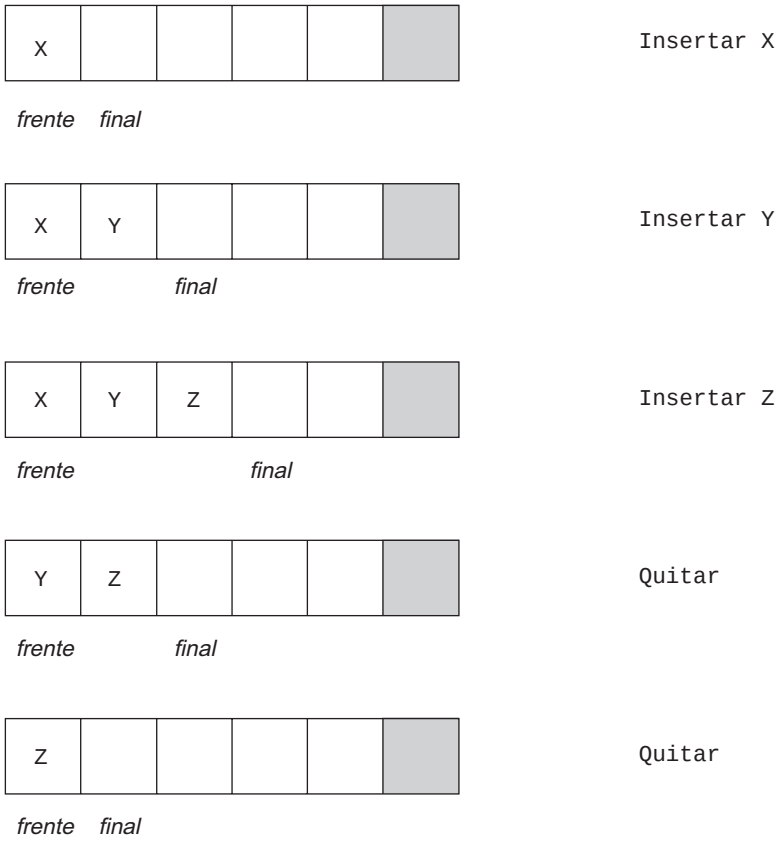


Figura 12.2. Oper aciones `Insertar` y `Quitar` en una `Cola`.

12.1.1. Especificaciones del tipo abstracto de datos `Cola`

Las operaciones que definen la estructura de una cola son las siguientes:

Tipo de dato	Elemento que se almacena en la cola.
Operaciones	
CrearCola	Inicia la cola como vacía.
Insertar	Añade un elemento por el final de la cola.
Quitar	Retira (extrae) el elemento frente de la cola.

Cola vacía	Comprobar si la cola no tiene elementos.
Cola llena	Comprobar si la cola está llena de elementos.
Frente	Obtiene el elemento frente o primero de la cola.
Tamaño de la cola	Número de elementos máximo que puede contener la cola.

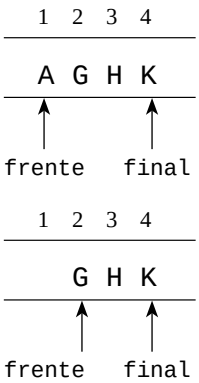
Desde el punto de vista de estructura de datos, una cola es similar a una pila, en cuanto que los datos se almacenan de modo lineal y el acceso a los datos sólo está permitido en los extremos de la cola.

La forma que los lenguajes tienen para representar el *TAD Cola* depende de donde se almacenen los elementos, en un array, en una estructura dinámica como puede ser una lista enlazada. La utilización de arrays tiene el problema de que la *cola* no puede crecer indefinidamente, está limitada por el tamaño del array, como contrapartida el acceso a los extremos es muy eficiente. Utilizar una lista dinámica permite que el número de nodos se ajuste al de elementos de la cola, por el contrario cada nodo necesita memoria extra para el enlace y también está el limite de memoria de la pila del computador.

12.2. COLAS IMPLEMENTADAS CON ARRAYS

Al igual que las pilas, las colas se implementan utilizando una estructura estática (arrays), o una estructura dinámica (listas enlazadas). La implementación estática se realiza declarando un array para almacenar los elementos, y dos marcadores o apuntadores para mantener las posiciones `frente` y `final` de la cola; es decir, un marcador apuntando a la posición de la *cabeza* de la cola y el otro al primer espacio vacío que sigue al *final* de la cola. Cuando un elemento se añade a la cola, se verifica si el marcador `final` apunta a una posición válida, entonces se asigna el elemento en esa posición y se incrementa el marcador `final` en 1. Cuando un elemento se elimina de la cola, se hace una prueba para ver si la cola está vacía y, si no es así, se recupera el elemento de la posición apuntada por el marcador (puntero) de cabeza y éste se incrementa en 1.

La operación de poner un elemento inserta por el extremo *final*. La primera asignación se realiza en la posición `final = 0`, cada vez que se añade un nuevo elemento se incrementa `final` en 1 y se asigna el elemento. La extracción de un elemento se hace por el extremo contrario, `frente`, cada vez que se extrae un elemento avanza `frente` una posición. La Figura 12.3 muestra el avance del puntero `frente` al extraer un elemento.



posición de `frente` y `final` después de extraer.

Figura 12.3. Una cola representada en un array.

El avance lineal de `frente` y `final` tiene un grave problema, deja *huecos* por la *izquierda del array*. Llegando a ocurrir que `final` alcance el índice más alto del array, no pudiéndose añadir nuevos elementos y, sin embargo, haya posiciones libres a la izquierda de `frente`.

Una alternativa que evita el problema de dejar *huecos*, consiste en mantener fijo el `frente` de la cola al comienzo del array, y mover todos los elementos de la cola una posición cada vez que se retira un elemento de la cola. Otra alternativa, mucho más eficiente, es considerar el array como una estructura *circular*.

12.2.1. Clase Cola

Los elementos de una cola pueden ser de cualquier tipo de dato: entero, cadena, objetos ...; por esa razón, se abstrae el tipo con la sentencia `typedef`, para que pueda sustituirse por cualquier tipo simple, posteriormente se implementa una *Cola Genérica* con *plantillas* (`template`).

La clase `ColaLineal` declara un array (`listaCola`) cuyo tamaño se determina por la constante `MAXTAMQ`. Las variables `frente` y `final` son los apuntadores a cabecera y cola, respectivamente. El constructor de la clase inicializa la estructura, de tal forma que se parte de una cola vacía.

Las operaciones básicas del tipo abstracto de datos cola: `insertar`, `quitar`, `colaVacia`, `colaLlena`, *y* `frente` se implementan en la clase. `insertar` toma un elemento y lo añade por el final. `quitar` suprime y devuelve el elemento *cabeza* de la cola. La operación `frente` devuelve el elemento que está en la primera posición (`frente`) de la cola, sin eliminar el elemento.

La operación de control, `colaVacia` comprueba si la cola tiene elementos, esta comprobación es necesaria antes de eliminar un elemento; `colaLlena` comprueba si se pueden añadir nuevos elementos, esta comprobación se realiza antes de insertar un nuevo miembro. Si las precondiciones para `insertar` y `quitar` se violan, el programa debe generar una excepción o error.

```
// archivo de cabecera ColaLineal.h

typedef tipo TipoDeDato;        // tipo ha de ser conocido
const int MAXTAMQ = 39;

class ColaLineal
{
protected:
     int frente;
     int final;
     TipoDeDato listaCola[MAXTAMQ];
public:
     ColaLineal()
     {
       frente = 0;
       final = -1;
     }
     // operaciones de modificación de la cola
     void insertar(const TipoDeDato& elemento)
```

```
        {
          if (!colaLlena())
          {
            listaCola[++final] = elemento;
          }
          else
            throw "Overflow en la cola";
        }
        TipoDeDato quitar()
        {
          if (!colaVacia())
          {
            return listaCola[frente++];
          }
          else
            throw "Cola vacia ";
        }
        void borrarCola()
        {
          frente = 0;
          final = -1;
        }
          // acceso a la cola
        TipoDeDato frenteCola()
        {
          if (!colaVacia())
          {
            return listaCola[frente];
          }
          else
            throw "Cola vacia ";
        }
          // métodos de verificación del estado de la cola
        bool colaVacia()
        {
          return frente > final;
        }
        bool colaLlena()
        {
          return final == MAXTAMQ - 1;
        }
};
```

Esta implementación de una cola es notablemente ineficiente, se puede alcanzar la condición de cola llena habiendo posiciones del array sin ocupar. Esto es debido a que al realizar la operación *quitar* avanza el `frente`, y, por consiguiente, las posiciones anteriores quedan desocupadas, no accesibles. Una solución a este problema consiste en que al retirar un elemento, `frente` no se incremente y se desplace el resto de elementos una posición a la izquierda.

Recodar

La realización de una cola con un arr ay lineal es notab lemente ineficiente, se puede alcanzar la condición de cola llena habiendo elementos del arr ay sin ocupar.

12.3. COLA CON UN ARRAY CIRCULAR

La alternativa, sugerida en la operación *quitar* un elemento, de desplazar los restantes elementos del array de modo que la *cabeza* de la cola vuelva al principio del array, es costosa, en términos de tiempo de computadora, especialmente si los datos almacenados en el array son estructuras de datos grandes.

La forma más eficiente de almacenar una cola en un array es modelar éste de tal forma que se una el extremo final con el extremo cabeza. Tal array se denomina *array circular* y permite que la totalidad de sus posiciones se utilicen para almacenar elementos de la cola sin necesidad de desplazar elementos. La Figura 12.4 muestra un *array circular* de *n* elementos.

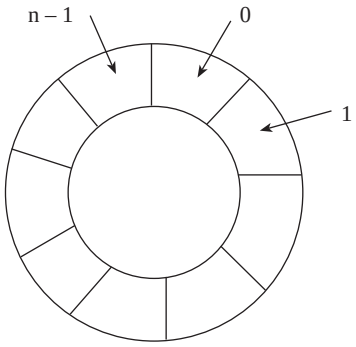


Figura 12.4. Un array circular.

El array se almacena de modo natural en la memoria, como un bloque lineal de *n* elementos. Se necesitan dos marcadores (apuntadores) *frente* y *final* para indicar, respectivamente, la posición del elemento *cabeza* y la posición donde se almacenó el último elemento puesto en la cola.

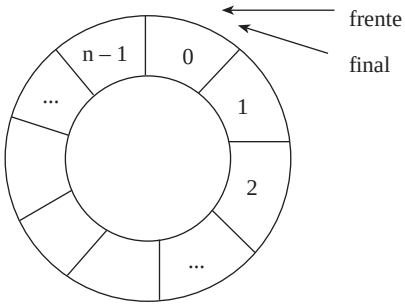


Figura 12.5. Una cola vacía.

El apuntador `frente` siempre contiene la posición del primer elemento de la cola y avanza en el sentido de las agujas del reloj; `final` contiene la posición donde se puso el último elemento, también avanza en el sentido del reloj (circularmente a la derecha). La implementación del movimiento circular se realiza segun la *teoría de los restos,* de tal forma que se generen índices de `0` a `MAXTAMQ - 1`:

Mover `final` adelante = `(final + 1) % MAXTAMQ`
Mover `frente` adelante = `(frente + 1) % MAXTAMQ`

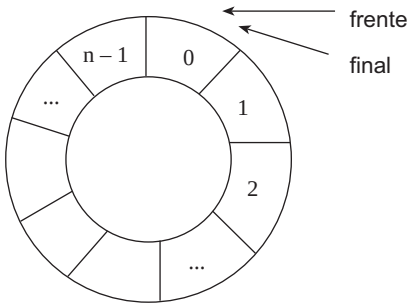


Figura 12.6. Una cola que contiene un elemento.

La implementación de la gestión de colas con un *array circular* ha de incluir las operaciones básicas del *TAD Cola*, en decir, las siguientes tareas básicas:

- Creación de una cola vacía, de tal forma que `final` apunte a una posición inmediatamente anterior a `frente`:

```
frente = 0; final = MAXTAMQ - 1.
```

- Comprobar si una cola está vacía:

```
frente == siguiente(final)
```

- Comprobar si una cola está llena. Para diferenciar la condición *cola llena* de *cola vacía* se sacrifica una posición del array , entonces la capacidad real de la cola será ser `MAXTAMQ-1`. La condición de cola llena:

```
frente == siguiente(siguiente(final))
```

- Poner un elemento a la cola: si la cola no está llena, fijar `final` a la siguiente posición: `final = (final + 1) % MAXTAMQ` y asignar el elemento.
- Retirar un elemento de la cola: si la cola no está vacía, quitarlo de la posición `frente` y establecer `frente` a la siguiente posición: `(frente + 1) % MAXTAMQ.`
- Obtener el elemento primero de la cola, si la cola no está vacía, sin suprimirlo de la cola.

12.3.1. Clase `Cola` con array circular

La representación de los elementos de la cola no cambia, un array lineal y dos índices: `listaCola[], frente, final.` Las operaciones tienen la misma interfaz que `ColaLineal`, entonces para aprovechar la potencia de la orientación a objetos, la nueva clase deriva de

`ColaLineal` y *redefine* las funciones de la *interfaz*. Además, se escribe la función auxiliar `siguiente()` para obtener la *siguiente* posición de una dada, aplicando la *teoría de los restos*.

A continuación, se codifica los métodos que implementan las operaciones del *TAD* `Cola`.

```
// archivo ColaCircular.h

#include "ColaLineal.h"
class ColaCircular : public ColaLineal
{

protected:
    int siguiente(int r)
    {
      return (r+1) % MAXTAMQ;
    }
    //Constructor, inicializa a cola vacía
public:
    ColaCircular()
    {
      frente = 0;
      final = MAXTAMQ-1;
    }
    // operaciones de modificación de la cola
    void insertar(const TipoDeDato& elemento);
    TipoDeDato quitar();
    void borrarCola();
    // acceso a la cola
    TipoDeDato frenteCola();
    // métodos de verificación del estado de la cola
    bool colaVacia();
    bool colaLlena();
};
```

Implementación

```
void ColaCircular :: insertar(const TipoDeDato& elemento)
{
  if (!colaLlena())
  {
    final = siguiente(final);
    listaCola[final] = elemento;
  }
  else
    throw "Overflow en la cola";
}

TipoDeDato ColaCircular :: quitar()
{
  if (!colaVacia())
  {
    TipoDeDato tm = listaCola[frente];
```

```
    frente = siguiente(frente);
    return tm;
  }
  else
    throw "Cola vacia ";
}

void ColaCircular :: borrarCola()
{
  frente = 0;
  final = MAXTAMQ-1;
}

TipoDeDato ColaCircular :: frenteCola()
{
  if (!colaVacia())
  {
    return listaCola[frente];
  }
  else
    throw "Cola vacia ";
}

bool ColaCircular :: colaVacia()
{
  return frente == siguiente(final);
}

bool ColaCircular :: colaLlena()
{
  return frente == siguiente(siguiente(final));
}
```

EJEMPLO 12.1. Se desea decidir si un número leído del dispositiv o estándar de entrada es capicúa.

El algoritmo para determinar si un número es capicúa utiliza conjuntamente una *Cola* y una *Pila*. El número se lee del teclado en forma de cadena de dígitos. La cadena se procesa carácter a carácter, es decir, dígito a dígito (un dígito es un carácter del '0' al '9'). Cada dígito se pone en la cola y a la vez en la pila. Una vez que se termina de leer los dígitos y de ponerlos en la cola y en la pila, comienza la comprobación: se extraen consecutivamente elementos de la cola y de la pila, y se comparan por igualdad, de producirse alguna no coincidencia entre dígitos es que el número no es capicúa y entonces se vacían las estructuras. El número es capicúa si el proceso de comprobación termina habiendo coincidido todos los dígitos en orden inverso, lo cual equivale a que la pila y la cola terminen vacías.

¿Por qué utilizar una pila y una cola?, sencillamente para asegurar que se procesan los dígitos en orden inverso; en la pila el *último en entrar es el primero en salir*, en la cola el *primero en entrar es el primero en salir*.

La pila que se implementa es de tipo `PilaGenérica` y la cola de la clase `ColaCircular` implementada con un *array circular*.

```
#include <iostream>
using namespace std;
```

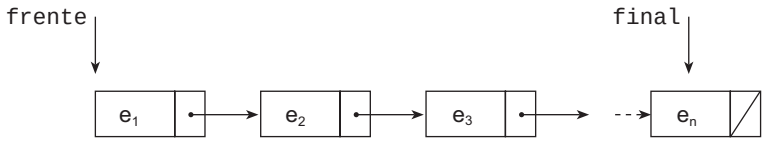
```
#include <string.h>
#include "PilaGenerica.h"
typedef char TipoDeDato;
#include "ColaCircular.h"

bool valido(const char* numero);

int main()
{
  bool capicua;
  char numero[81];
  PilaGenerica<char> pila;
  ColaCircular q;

  capicua = false;
  while (!capicua)
  {
    do {
      cout << "\nTeclea el número: ";
      cin.getline(numero,80);
    }while (!valido(numero)); // todos los caracteres dígitos
    // pone en la cola y en la pila cada dígito
    for (int i = 0; i < strlen(numero); i++)
    {
      char c;
      c = numero[i];
      q.insertar(c);
      pila.insertar(c);
    }
    // se retira de la cola y la pila para comparar
    do {
      char d;
      d = q.quitar();
      capicua = d == pila.quitar();    //compara por igualdad
    } while (capicua && !q.colaVacia());

    if (capicua)
      cout << numero << " es capicúa " << endl;
    else
    {
      cout << numero << " no es capicúa, ";
      cout << " intente con otro. ";
     // se vacía la cola y la pila
      q.borrarCola();
      pila.limpiarPila();
    }
  }
  return 0;
}

    // verifica que cada carácter es dígito
bool valido(const char* numero)
{
  bool sw = true;
  int i = -1;
```

```
    while (sw && (i < strlen(numero)))
    {
      char c;
      c = numero[++i];
      sw = (c >= '0' && c <= '9');
    }
    return sw;
  }
```

12.4. COLA GENÉRICA CON UNA LISTA ENLAZADA

La implementación del *TAD* `Cola` con independencia del tipo de dato de los elementos se consigue utilizando las *plantillas* (`template`) de C++. Además, se va a utilizar la estructura dinámica lista enlazada para que, en todo momento, el número de elementos de la cola se ajuste al número de nodos de la lista, de tal forma que pueda crecer o disminuir sin problemas de espacio.

La implementación del *TAD* `Cola` con una lista enlazada utiliza dos punteros de acceso a la lista: `frente` y `final`. Son los extremos por donde salen y por donde se ponen, respectivamente, los elementos de la cola.



e_1, e_2, ...en son valores del tipo *TipoDeDato*

Figura 12.9. Cola con lista enlazada (representación gráfica típica).

El puntero `frente` apunta al primer elemento de la lista y, por tanto, de la cola (el primero en ser retirado), `final` apunta al último elemento de la lista y también de la cola.

La lista enlazada crece y decrece según las necesidades, según se incorporen elementos o se retiren, y por esta razón en esta implementación no se considera la función de control de `Colallena`.

12.4.1. Clase genérica `Cola`

La implementación de una *cola genérica* se realiza con dos clases: clase `ColaGenerica` y clase `Nodo` que será una clase anidada. El `Nodo` representa al elemento y al enlace con el siguiente nodo; al crear un `Nodo` se asigna el elemento y el enlace se pone `NULL`.

La clase `ColaGenerica` define las variables de acceso: `frente` y `final`, y las operaciones básicas del *TAD* `Cola`. Su constructor inicializa `frente` y `final` a `NULL`, es decir, a la condición *cola vacía*.

```
  // archivo ColaGenerica.h
template <class T>
class ColaGenerica
```

```
{
protected:
    class NodoCola
    {
    public:
         NodoCola* siguiente;
         T elemento;
         NodoCola (T x)
         {
            elemento = x;
            siguiente = NULL;
         }
    };
    NodoCola* frente;
    NodoCola* final;

public:
    ColaGenerica()
    {
      frente = final = NULL;
    }
    void insertar(T elemento);
    T quitar();
    void borrarCola();
    T frenteCola()const;
    bool colaVacia() const;
    ~ColaGenerica()
    {
      borrarCola ();
    }
};
```

12.4.2. Implementación de las operaciones de cola genérica

Las operaciones acceder directamente a la lista; `insertar` crea un nodo y lo enlaza por el `final`, `quitar` devuelve el dato del nodo `frente` y lo borra de la lista, `frenteCola` accede al `frente` para obtener el elemento.

Añadir un elemento a la cola

```
template <class T>
void ColaGenerica<T> :: insertar(T elemento)
{
  NodoCola* nuevo;
  nuevo = new NodoCola (elemento);
  if (colaVacia())
  {
    frente = nuevo;
  }
  else
  {
    final -> siguiente = nuevo;
  }
  final = nuevo;
}
```

Sacar de la cola

Devuelve el elemento frente y lo quita de la cola, disminuye el tamaño de la cola.

```
template <class T>
T ColaGenerica<T> :: quitar()
{
  if (colaVacia())
    throw "Cola vacía, no se puede extraer.";
  T aux = frente -> elemento;
  NodoCola* a = frente;
  frente = frente -> siguiente;
  delete a;
  return aux;
}
```

Elemento frente de la cola

```
template <class T>
T ColaGenerica<T> :: frenteCola()const
{
  if (colaVacia())
    throw "Cola vacía";
  return frente -> elemento;
}
```

Vaciado de la cola

Elimina todos los elementos de la cola. Recorre la lista desde `frente` a `final`, es una operación de complejidad lineal, `O(n)`.

```
template <class T>
void ColaGenerica<T> :: borrarCola()
{
  for (;frente != NULL;)
  {
      NodoCola* a;
      a = frente;
      frente = frente -> siguiente;
      delete   a;
  }
  final = NULL;
}
```

Verificación del estado de la cola

```
template <class T>
bool ColaGenerica<T> :: colaVacia() const
{
  return frente == NULL;
}
```

EJERCICIO 12.1. Se quiere generar números de la suerte aplicando una variación al llamado "problema de José". El punto de partida es una lista de n números, esta lista se va reduciendo siguiendo el siguiente algoritmo:

1. *Se genera un número aleatorio* `n1`.
2. *Si* `n1 <= n` *se quitan de la lista los números que ocupan las posiciones* `1, 1 + n1, 1 + 2 * n1, ...; n` *toma el valor del número de elementos que quedan en la lista.*
3. *Se vuelve al paso 1.*
4. *Si* `n1 > n` *fin del algoritmo, los números de la suerte son los que quedan en la lista.*

El problema se va a resolver utilizando una *cola*. En primer lugar, se genera la lista de `n` números aleatorios que se almacenan en la cola. A continuación, se siguen los pasos del algoritmo, en cada pasada se eliminan los elementos de la cola que están en las posiciones *`(múltiplos de n1) + 1`*. Estas posiciones, denominadas `i`, se pueden expresar matemáticamente:

```
i modulo n1 == 1
```

Una vez que termina el algoritmo, los números de la suerte son los que han quedado en la cola, entonces se retiran de la cola y se escriben.

Se utiliza una *cola genérica*, implementada con listas enlazadas. Al crear la cola se pasa el argumento `int` que, en este ejercicio, es el tipo de dato de los elementos.

```
#include <iostream>
using namespace std;
#include <time.h>
#include "ColaGenerica.h"

const int N = 99;
#define randomize (srand(time(NULL)))
#define random(num) (rand()%(num))

template <class T>
void mostrarCola(ColaGenerica<T>& q);

int main()
{
  int n, n1, n2, i;
  ColaGenerica<int> q;

  randomize;
  // número inicial de elementos de la lista
  n = 11 + random(N);
  // se generan n números aleatorios
  for (int i = 1; i <= n; i++)
  {
    q.insertar(random(N * 3));
  }
```

```
  // se genera aleatoriamente el intervalo n1
  n1 = 1 + random(11);
  // se retiran de la cola elementos a distancia n1
  while (n1 <= n)
  {
    int nt;
    n2 = 0; // contador de elementos que quedan
    for (i = 1; i <= n; i++)
    {
      nt = q.quitar();
      if (i % n1 == 1)
      {
        cout << "\n Se quita " << nt << endl;
      }
      else
      {
        q.insertar(nt); // se vuelve a meter en la cola
        n2++;
      }
    }
    n = n2;
    n1 = 1 + random(11);
  }
  cout << "\n\t Los números de la suerte: ";
  mostrarCola(q);
  return 0;
}

template <class T>
void mostrarCola(ColaGenerica<T>& q)
{
  while (! q.colaVacia())
  {
    int v;
    v = q.quitar();
    cout << v << " ";
  }
  cout << endl;
}
```

12.5. BICOLAS: COLAS DE DOBLE ENTRADA

Una *bicola* o *cola de doble entrada* es un conjunto *ordenado* de elementos al que se puede *añadir* o *quitar* elementos desde cualquier extremo del mismo. El acceso a la bicola está permitido desde cualquier extremo, por lo que se considera que es una *cola bidireccional*. La estructura *bicola* es una extensión del *TAD Cola*.

Los dos extremos de una bicola se identifican con los apuntadores `frente` y `final` (mismos nombres que en una cola). Las operaciones básicas que definen una bicola son una ampliación de la operaciones de una cola :

CrearBicola : inicializa una bicola sin elementos.
BicolaVacia : devuelve `true` si la bicola no tiene elementos.

PonerFrente : añade un elemento por extremo frente.
PonerFinal : añade un elemento por extremo final.
QuitarFrente : devuelve el elemento `frente` y lo retira de la bicola.
QuitarFinal : devuelve el elemento `final` y lo retira de la bicola.
Frente : devuelve el elemento `frente` de la bicola.
Final : devuelve el elemento `final` de la bicola.

Al tipo de datos bicola se puede poner restricciones respecto a la entrada o a la salida de elementos. Una *bicola con restricción de entrada* es aquella que sólo permite inserciones por uno de los dos extremos, pero que permite retirar elementos por los dos extremos. Una *bicola con restricción de salida* es aquella que permite inserciones por los dos extremos, pero sólo permite retirar elementos por un extremo.

La representación de una bicola puede ser con un array, con un *array circular*, o bien con *listas enlazadas*. Siempre se debe disponer de dos *marcadores* o variables índice (*apuntadores*) que se correspondan con los extremos, *frente* y *final*, de la estructura.

12.5.1. Bicola genérica con listas enlazadas

La implementación del *TAD* `Bicola` con una lista enlazada se caracteriza por ajustarse al número de elementos; es una implementación dinámica, *crece* o *decrece* según lo requiera la ejecución del programa que utiliza la bicola. Como los elementos de una bicola, y en general de cualquier *estructura contenedora*, pueden ser de cualquier tipo, se declara la clase genérica `Bicola`. Además, la clase va a heredar de `ColaGenerica` ya que es una extensión de ésta.

```
template <class T> class BicolaGenerica : public ColaGenerica<T>
```

De esta forma, `BicolaGenerica` dispone de todas las funciones y atributos de la clase `ColaGenerica`. Entonces, sólo es necesario codificar las operaciones de `Bicola` que no están implementadas en `ColaGenerica`.

```
// archivo BicolaGenerica.h
#include "ColaGenerica.h"

template <class T>
class BicolaGenerica : public ColaGenerica<T>
{
public:
  void ponerFinal(T elemento);
  void ponerFrente(T elemento);
  T quitarFrente();
  T quitarFinal();
  T frenteBicola()const;
  T finalBicola() const;
  bool bicolaVacia();
  void borrarBicola();
  int numElemsBicola() const; // cuenta los elementos de la bicola
};
```

12.5.2. Implementación de las operaciones de `BicolaGenerica`

Las funciones: `ponerFinal(), quitarFrente(), bicolaVacia(), frenteBicola()` son identicas a las funciones de la clase `ColaGenerica insertar(), quitar(), colaVacia() y frenteCola()` respectivamente, y como por el mecanismo de la derivación de clases se han heredado, su implementación consiste en una simple llamada a la correspondiente función heredada.

Añadir un elemento a la bicola

Añadir por el extremo final de `Bicola`.

```
template <class T>
void BicolaGenerica<T> :: ponerFinal(T elemento)
{
  insertar(elemento); // heredado de ColaGenerica
}
```

Añadir por el extremo `frente` de `Bicola`.

```
template <class T>
void BicolaGenerica<T> :: ponerFrente(T elemento)
{
  NodoCola* nuevo;

  nuevo = new NodoCola(elemento);
  if (bicolaVacia())
  {
    final = nuevo;
  }
  nuevo -> siguiente = frente;
  frente = nuevo;
}
```

Sacar un elemento de la bicola

Devuelve el elemento `frente` y lo quita de la *Bicola*, disminuye su tamaño.

```
template <class T>
T BicolaGenerica<T> :: quitarFrente()
{
  return quitar();  // método heredado de ColaLista
}
```

Devuelve el elemento `final` y lo quita de la *Bicola*, disminuye su tamaño. Es necesario recorrer la lista para situarse en el nodo anterior a `final`, y después enlazar.

```
template <class T>
T BicolaGenerica<T> :: quitarFinal()
{
  T aux;
  if (! bicolaVacia())
```

```
  {
    if (frente == final)   // Bicola dispone de un solo nodo
    {
      aux = quitar();
    }
    else
    {
      NodoCola* a = frente;
      while (a -> siguiente != final)
         a = a -> siguiente;
      aux = final -> elemento;
      final = a;
      delete (a -> siguiente);
    }
  }
  else
    throw "Eliminar de una bicola vacía";
  return aux;
}
```

Acceso a los extremos de la bicola

Elemento frente

```
template <class T>
T BicolaGenerica<T>:: frenteBicola()const
{
  return frenteCola();   // heredado de ColaGenerica
}
```

Elemento final

```
template <class T>
T BicolaGenerica<T>:: finalBicola() const
{
  if (bicolaVacia())
  {
    throw "Error: bicola vacía";
  }
  return (final -> elemento);
}
```

Vaciado de la bicola

```
template <class T>
void BicolaGenerica<T> :: borrarBicola()
{
  borrarCola();        // heredado de ColaGenerica
}
```

Verificación del estado y número de elementos de la bicola

```
template <class T>
bool BicolaGenerica<T> :: bicolaVacia() const
{
  return colaVacia();  // heredado de ColaGenerica
}
```

La siguiente operación recorre la estructura, de `frente` a `final`, para contar el número de elementos de que consta.

```
template <class T>
int BicolaGenerica<T> :: numElemsBicola() const
{
  int n = 0;
  NodoCola* a = frente;
  if (!bicolaVacia())
  {
    n = 1;
    while (a != final)
    {
      n++;
      a = a -> siguiente;
    }
  }
  return n;
}
```

EJERCICIO 12.2. La salida a pista de las a vionetas de un aeródromo está organizada en forma de fila(línea), con una capacidad máxima de apar atos en esper a de 16 a vionetas. Las avionetas llegan por el e xtremo izquierdo (final) y salen por el e xtremo derecho (frente). Un piloto puede decidir retir arse de la fila por r azones técnicas, en ese caso todas las a vionetas que siguen han de ser quitadas de la fila, retir ar el apar ato y las a vionetas desplazadas colocarlas de n uevo en el mismo orden relativ o en que estaban. La salida de una a vioneta de la fila supone que las demás son mo vidas hacia adelante , de tal f orma que los espacios libres del estacionamiento estén en la par te izquierda (final).

La aplicación para emular este estacionamiento tiene como entrada un carácter que indica una acción sobre la avioneta, y la matrícula de la avioneta. La acción puede ser llegada (E), salida (S) de la avioneta que ocupa la primera posición y retirada (T) de una avioneta de la fila. En la llegada puede ocurrir que el estacionamiento esté lleno, si esto ocurre la avioneta espera hasta que se quede una plaza libre.

El estacionamiento va a estar representado por una *bicola*, (realmente debería ser una *bicola* de salida restringida). ¿Por qué esta elección?, la salida siempre se hace por el mismo extremo, sin embargo la entrada se puede hacer por los dos extremos, y así se contempla dos acciones: *llegada* de una avioneta nueva; y *entrada de una avioneta que ha sido movida* para que salga una intermedia.

Las avionetas que se mueven para poder retirar del estacionamiento una intermedia, se disponen en una *pila*, así la ultima en entrar será la primera en añadirse al extremo salida del estacionamiento (*bicola*) y seguir en el mismo orden relativo.

Las avionetas se representan mediante una cadena para almacenar, simplemente, el número de matrícula. Entonces, los elementos de la pila y de la *bicola* son de tipo *cadena* (`string`).

Se utiliza la implementación de `PilaGenerica`, en esta ocasión el tipo de dato de los elementos es `string`. La bicola utilizada es de tipo `BicolaGenerica`, también el tipo de dato de los elementos es `string`.

La función `main()` gestiona las operaciones indicadas en ejercicio. La resolución del problema no toma acción cuando una avioneta no puede incorporarse a la fila por estar llena. El

lector puede añadir el código necesario para que, utilizando una cola, las avionetas que no pueden entrar en la fila se guarden en la cola, y cada vez que salga una avioneta se añada otra desde la cola.

En la función `retirar()` se simula el hecho de que una avioneta, que se encuentra en cualquier posición de la *bicola*, decide salir de la fila; la función retira de la fila las avionetas por el *frente*, a la vez que las guarda en una pila, hasta que encuentra la avioneta a retirar. A continuación, se insertan en la fila, por el *frente*, las avionetas de la pila, así quedan en el mismo orden que estaban anteriormente. La constante `maxAvtaFila (16)` guarda el número máximo de avionetas que pueden estar en la fila esperando la salida.

```
#include <iostream>
using namespace std;
#include <string>
#include <ctype.h>
#include "BicolaGenerica.h"
#include "PilaGenerica.h"

const int maxAvtaFila = 16;
template <class T>
bool retirar(BicolaGenerica<T>& fila, string avioneta);

int main()
{
  string avta;
  char ch;
  BicolaGenerica<string> fila;
  bool esta, mas = true;

  while (mas)
  {
    char bf[81];
    cout << "Entrada: acción(E/S/T)matrícula."
         << " Para terminar la simulación: X." << endl;
    do {
      cin >> ch; ch = toupper(ch); cin.ignore(2,'\n');
    } while(ch != 'E' && ch != 'S' && ch != 'T' && ch != 'X');
    if (ch == 'S')              // sale de la fila una avioneta
    {
      if (!fila.bicolaVacia())
      {
        avta = fila.quitarFrente();
        cout << "Salida de la avioneta: " << avta <<endl;
      }
    }
    else if (ch == 'E') // llega a la fila una avioneta
    {
      if (fila.numElemsBicola() < maxAvtaFila)
      {
        cout << " Matricula avioneta: " <<endl;
        cin.getline(bf,80);
        avta = bf;
        fila.ponerFinal(avta);
      }
    }
```

```
      else if (ch == 'T')  // avioneta abandona la fila
      {
        cout << " Matricula avioneta: " << endl;
        cin.getline(bf,80);
        avta = bf;
        esta = retirar(fila, avta);
        if (!esta)
           cout << "¡¡ avioneta no encontrada !!" <<endl;
      }
      mas = !(ch == 'X');
    }
    return 0;
  }

  template <class T>
  bool retirar(BicolaGenerica<T>& fila, string avioneta)
  {
    bool encontrada = false;
    PilaGenerica<string> pila;
    while (!encontrada && !fila.bicolaVacia())
    {
      string avta;
      avta = fila.quitarFrente();
      if (avioneta == avta) // sobrecarga del operator ==
         { encontrada = true;
           cout << "Avioneta" <<avta << "retirada" <<endl;
         }
      else pila.insertar (avta);
    }
    while (!pila.pilaVacia())
        fila.ponerFrente (pila.quitar());
    return encontrada;
  }
```

RESUMEN

Una cola es una lista lineal en la que los datos se insertan por un extremo (`final`) y se extraen por el otro extremo (`frente`). Es una estructura *FIFO* (*first in first out*, primero en entrar primero en salir).

Las operaciones básicas que se aplican sobre colas: `crearCola, colaVacia, colaLlena, insertar, frente` y `quitar`.

`crearCola`, inicializa a una cola sin elementos. Es la primera operación a realizar con una cola. La operación queda implementada en el constructor de la clase `Cola`.

`colaVacia`, determina si una cola tiene o no elementos. Devuelve `true` si no tiene elementos.

`colaLlena`, determina si no se pueden almacenar más elementos en una cola. Se aplica esta operación cuando se utiliza un array para guardar los elementos de la cola.

`insertar`, añade un nuevo elemento a la cola, siempre por el extremo `final`.

`frente`, devuelve el elemento que está, justamente en el extremo `frente` de la cola, sin extraerlo.

`quitar`, extrae el elemento frente de la cola y lo elimina.

La implementación del *TAD Cola*, en C++, se realiza con arrays, o bien con listas enlazadas. La implementación con un array lineal es muy ineficiente; se ha de considerar el array como una estructura circular y aplicar la teoría de los restos para avanzar el *frente* y el *final* de la cola.

La realización de una cola con listas enlazadas permite que el tamaño de la estructura se ajuste al número de elementos. La cola puede crecer indefinidamente, con el único tope de la memoria libre.

Las colas se utilizan en numerosos modelos de sistemas del mundo real: cola de impresión en un servidor de impresoras, programas de simulación, colas de prioridades en organización de viajes.

Las *bicolas* son *colas dobles* en el sentido de que las operaciones básicas *insertar* y *retirar* elementos se realizan por los dos extremos. A veces se ponen restricciones de entrada o de salida por algún extremo. Una bicola es, realmente, una extensión de una cola. La implementación natural del *TAD Bicola* es con una clase derivada de la clase `Cola`.

EJERCICIOS

12.1. Considerar una cola de nombres representada por una array circular con 6 posiciones, el campo frente con el valor: `frente = 2`. Y los elementos de la cola: `Mar, Sella, Centurión`.
Escribir los elementos de la cola y los campos `frente` y `final` según se realizan estas operaciones:

- Añadir `Gloria` y `Generosa` a la cola.
- Eliminar de la cola.
- Añadir `Positivo`.
- Añadir `Horche`.
- Eliminar todos los elementos de la cola.

12.2. A la clase que representa una cola implementada con un array circular y dos variables `frente` y `final`, se le añade una variable más que guarda el número de elementos de la cola. Escribir de nuevo las funciones de manejo de colas considerando este campo contador.

12.3. Suponer que para representar una `bicola` se ha elegido una lista doblemente enlazada, y que los extremos de la lista se denominan `frente` y `final`. Escribir la clase *Bicola* con esta representación de los datos y las operaciones del *TAD Bicola*.

12.4. Supóngase que se tiene la clase `Cola` que implementa las operaciones del *TAD Cola*. Escribir una función para crear *un clon* (una copia) de una cola determinada. Las operaciones que se han de utilizar serán únicamente las del *TAD Cola*.

12.5. Considere una *bicola* de caracteres, representada en un array circular. El array consta de 9 posiciones. Los extremos actuales y los elementos de la bicola:

```
frente = 5 final = 7   bicola: A,C,E
```

Escribir los extremos y los elementos de la bicola según se realizan estas operaciones:

- Añadir los elementos `F` y `K` por el `final` de la bicola.
- Añadir los elementos `R`, `W` y `V` por el `frente` de la bicola.

- Añadir el elemento `M` por el `final` de la bicola.
- Eliminar dos caracteres por el `frente`.
- Añadir los elementos `K` y `L` por el `final` de la bicola.
- Añadir el elemento `S` por el `frente` de la bicola.

12.6. Se tiene una pila de enteros positivos. Con las operaciones básicas de pilas y colas escribir un fragmento de código para poner todos los elementos de la pila que sean pares en la cola.

12.7. Implementar el *TAD Cola* utilizando una lista enlazada circular. Por conveniencia, establecer el acceso a la lista, `lc`, por el último nodo (elemento) insertado, y considerar al nodo siguiente de `lc` el primero o el que más tarde se insertó.

PROBLEMAS

12.1. Con un archivo de texto se quieren realizar las siguientes acciones: formar una lista de colas, de tal forma que cada nodo de la lista esté en la dirección de una cola que tiene todas las palabras del archivo que empiezan por una misma letra. Visualizar las palabras del archivo, empezando por la cola que contiene las palabras que comienzan por *a*, a continuación las de la letra *b*, y así sucesivamente.

12.2. Se tiene un archivo de texto del cual se quiere determinar las frases que son palíndromo. Para lo cual se ha de seguir la siguiente estrategia:

- Considerar cada línea del texto una frase.
- Añadir cada carácter de la frase a una pila y a la vez a una cola.
- Extraer carácter a carácter, y simultáneamente de la pila y de la cola. Su comparación determina si es palíndromo o no.

Escribir un programa que lea cada línea del archivo y determine si es palíndromo.

12.3. Escribir un programa en el que se generen 100 números aleatorios en el rango `-25 .. +25` y se guarden en una cola implementada mediante un array circular. Una vez creada la cola, el usuario puede pedir que se forme otra cola con los números negativos que tiene la cola original.

12.4. Escribir una función que tenga como argumentos dos colas del mismo tipo. Devuelva cierto si las dos colas son idénticas.

12.5. Un pequeño supermercado dispone en la salida de tres cajas de pago. En el local hay 25 carritos de compra. Escribir un programa que simule el funcionamiento del supermercado, siguiendo las siguientes reglas:

- Si cuando llega un cliente no hay ningún carrito disponible, espera a que lo haya.
- Ningún cliente se impacienta y abandona el supermercado sin pasar por alguna de las colas de las cajas.
- Cuando un cliente finaliza su compra, se coloca en la cola de la caja que hay menos gente, y no se cambia de cola.
- En el momento en el cual un cliente paga en la caja, el carro de la compra que tiene queda disponible.

Representar la lista de carritos de la compra y las cajas de salida mediante colas.

12.6. En un archivo F están almacenados números enteros arbitrariamente grandes. La disposición es tal que hay un número entero por cada línea de F. Escribir un programa que muestre por pantalla la suma de todos los números enteros. Al resolver el problema habrá que tener en cuenta que al ser enteros grandes no pueden almacenarse en variables numéricas.
Utilizar dos pilas para guardar los dos primeros números enteros, almacenándose dígito a dígito. Al extraer los elementos de la pila salen en orden inverso y, por tanto, de menos peso a mayor peso, se suman dígito con dígito y el resultado se guarda en una cola, también dígito a dígito. A partir de este primer paso, se obtiene el siguiente número del archivo, se guarda en una pila y a continuación se suma dígito a dígito con el número que se encuentra en la cola; el resultado se guarda en otra cola. El proceso se repite, nuevo número del archivo se mete en la pila, que se suma con el número actual de la cola.

12.7. Una empresa de reparto de propaganda contrata a sus trabajadores por días. Cada repartidor puede trabajar varios días continuados o alternos. Los datos de los repartidores se almacenan en una lista enlazada. El programa a desarrollar contempla los siguientes puntos:

- Crear una cola que guarde el número de la seguridad social de cada repartidor y la entidad anunciada en la propaganda para un único día de trabajo.
- Actualizar la lista citada anteriormente (que ya existe con contenido) a partir de los datos de la cola.

La información de la lista es la siguiente: número de seguridad social, nombre y total de días trabajados. Además, está ordenada por el número de la seguridad social. Si el trabajador no está incluido en la lista debe añadirse a la misma de tal manera que siga ordenada.

12.8. El supermercado "Esperanza" quiere simular los tiempos de atención al cliente a la hora de pasar por la caja, los supuestos de que se parte para la simulación son los siguientes:

- Los clientes forman una única fila. Si alguna caja está libre, el primer cliente de la fila es atendido. En el caso de que haya más de un caja libre, la elección del número de caja por parte del cliente es aleatoria.
- El número de cajas de que se dispone para atención a los clientes es de tres, salvo que haya mas de 20 personas esperando en la fila, entonces se habilita una cuarta caja, ésta se cierra cuando no quedan clientes esperando. El tiempo de atención de cada una de las caja está distribuido uniformemente, la caja 1 entre 1.5 y 2.5 minutos; la caja 2 entre 2 y 5 minutos, la caja 3 entre 2 y 4 minutos. La caja 4, cuando está abierta tiene un tiempo de atención entre 2 y 4.5 minutos.
- Los clientes llegan a la salida en intervalos de tiempo distribuidos uniformemente, con un tiempo medio de 1 minuto.

El programa de simulación se pide hacerlo para 7 horas de trabajo. Se desea obtener una estadística con los siguientes datos:

- Clientes atendidos durante la simulación.
- Tamaño medio de la fila de clientes.
- Tamaño máximo de la fila de clientes.
- Tiempo máximo de espera de los clientes.
- Tiempo en que está abierto la cuarta caja.

12.9. La institución académica de *La Alcarria* dispone de 15 computadores conectados a Internet. Se desea hacer una simulación de la utilización de los computadores por los alumnos. Para

ello se supone que la frecuencia de llegada de un alumno es de 18 minutos las 2 dos primeras horas y de 15 minutos el resto del día. El tiempo de utilización del computador es un valor aleatorio, entre 30 y 55 minutos.
El programa debe tener como salida líneas en las que se refleja la llegada de un alumno, la hora en que llega y el tiempo de la conexión. En el supuesto de que llegue un alumno y no existan computadores libres el alumno no espera, se mostrará el correspondiente aviso. En una cola de prioridad se tiene que guardar los distintos "eventos" que se producen, de tal forma que el programa avance de evento a evento. Suponer que la duración de la simulación es de las 10 de la mañana a las 8 de la tarde.

12.10. La entrada a una sala de arte que ha inaugurado una gran exposición sobre la evolución del arte rural, se realiza por tres torniquetes. Las personas que quieren ver la exposición forman un única fila y llegan de acuerdo a una distribución exponencial, con un tiempo medio entre llegadas de 2 minutos. Una persona que llega a la fila y ve más de 10 personas esperando se va con una probabilidad del 20 por 100, aumentando en 10 puntos por cada 15 personas más que haya esperando, hasta un tope del 50 por 100. El tiempo medio que tarda una persona en pasar es de 1 minuto (compra de la entrada y revisión de los bolsos). Además, cada visitante emplea en recorrer la exposición entre 15 y 25 minutos distribuido uniformemente. La sala sólo admite, como máximo, 50 personas. Simular el sistema durante un período de 6 horas para determinar:

- Número de personas que llegan a la sala y número de personas que entran.
- Tiempo medio que debe esperar una persona para entrar en la sala.