## 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.

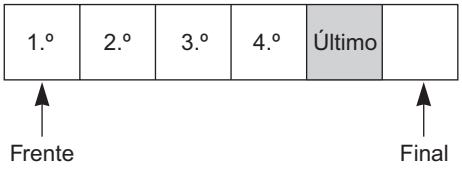



**Figura 12.1.** Una cola.

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.

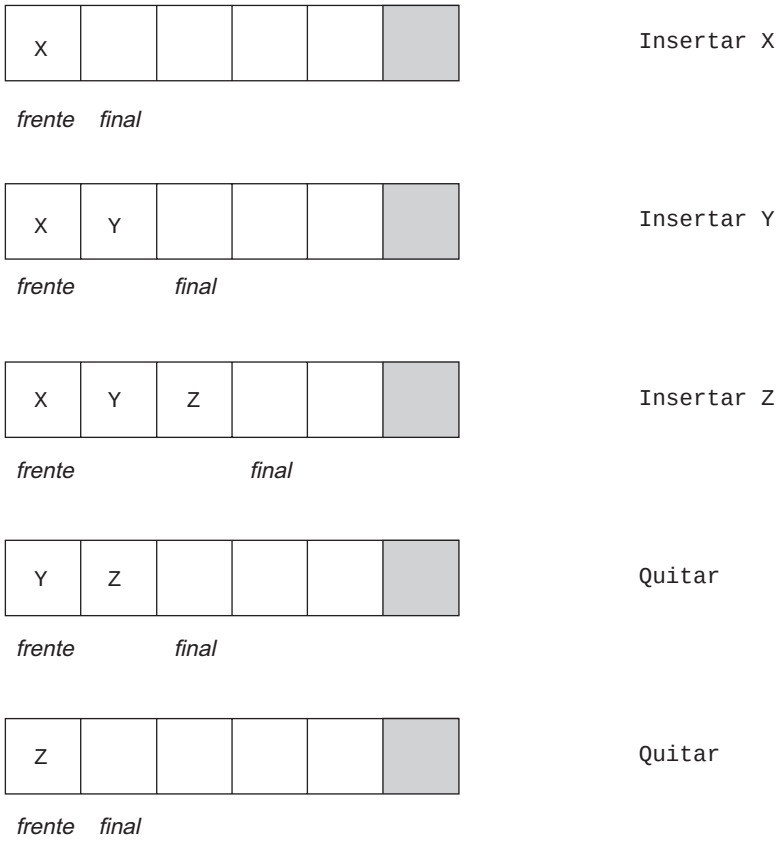



**Figura 12.2.** Oper aciones `Insertar` y `Quitar` en una `Cola`.

### 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. |

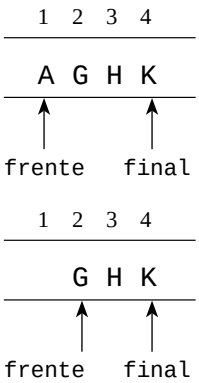
Desde el punto de vista de estructura de datos, una cola es similar a una pila, en cuanto que los datos se almacenan de modo lineal y el acceso a los datos sólo está permitido en los extremos de la cola.

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.

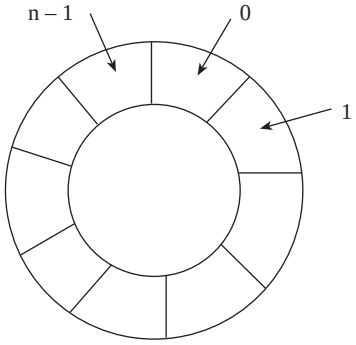



**Figura 12.4.** Un array circular.

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.

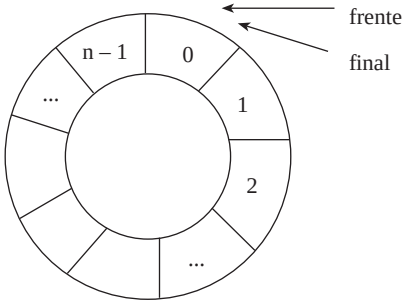



**Figura 12.5.** Una cola vacía.

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`

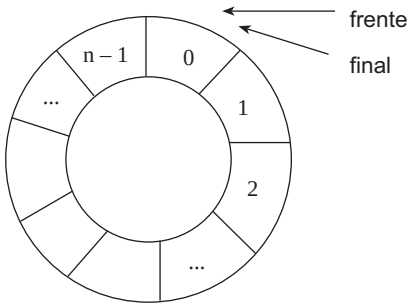



**Figura 12.6.** Una cola que contiene un elemento.

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;
```

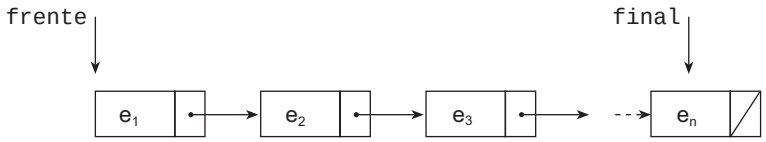
```
#include <string.h>
#include "PilaGenerica.h"
typedef char TipoDeDato;
#include "ColaCircular.h"

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.