12. COLAS

12.0 Introducción

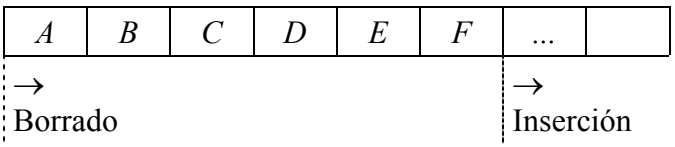
En este tema veremos las estructuras de datos lineales colas, su significado y las operaciones más habituales sobre ellas, así como algunas posibles implementaciones con estructuras estáticas y dinámicas en C++.

12.1 Fundamentos

Las colas son secuencias de elementos caracterizadas porque las operaciones de inserción y borrado se realizan sobre extremos opuestos de la secuencia. La inserción se produce en el "final" de la secuencia, mientras que el borrado se realiza en el otro extremo, el "inicio" de la secuencia.

Las restricciones definidas para una cola hacen que el primer elemento que se inserta en ella sea, igualmente, el primero en ser extraido de la estructura. Si una serie de elementos A, B, C, D, E se insertan en una cola en ese mismo orden, entonces los elementos irán saliendo de la cola en el ordenen que entraron. Por esa razón, en ocasiones, las colas se conocen con el nombre de listas o secuencias FIFO (*First In First Out*, el primero que entra es el primero que sale).

Estructura Cola



Las colas, al igual que las pilas, resultan de aplicación habitual en muchos problemas informáticos. Quizás la aplicación más común de las colas es la organización de tareas de un ordenador. En general, los trabajos enviados a un ordenador son "encolados" por éste, para ir procesando secuencialmente todos los trabajos en el mismo orden en que se reciben. Cuando el ordenador recibe el encargo de realizar una tarea, ésta es almacenada al final de la cola de trabajos. En el momento que la tarea que estaba realizando el procesador acaba, éste selecciona la tarea situada al principio de la cola para ser ejecutada a continuación. Todo esto suponiendo la ausencia de prioridades en los trabajos. En caso contrario, existirá una cola para cada prioridad. Del mismo modo, es necesaria una cola, por ejemplo, a la hora de gestionar eficientemente los trabajos que deben ser enviados a una impresora (o a casi cualquier dispositvo conectado a un ordenador). De esta manera, el ordenador controla el envio de trabajos al dispositivo, no enviando un trabajo hasta que la impresora no termine con el anterior.

Análogamente a las pilas, es necesario definir el conjunto de operaciones básicas para especificar adecuadamente una estructura cola. Estas operaciones serían:

- Crear una cola vacía.
- Determinar si la cola está vacía, en cuyo caso no es posible eliminar elementos.
- Acceder al elemento inicial de la cola.
- Insertar elementos al final de la cola.
- Eliminar elementos del inicio de la cola.

Al igual que realizamos con las pilas, haremos una declaración un poco más formal de estas operaciones y los axiomas que las caracterizan:

Estructura

Cola (Valor) *{* Valor será el tipo de datos que podremos guardar en la cola *}*

Operaciones

CREAR_COLA ( ) → Cola
ENCOLAR (Cola , Valor) → Cola
DESENCOLAR (Cola) → Cola
PRIMERO_COLA (Cola) → Valor
COLA_VACIA (Cola) → Lógico

Axiomas

$\forall$ *queue* $\in$ *Cola, x* $\in$ *Valor se cumple que:*

COLA_VACIA (CREAR_COLA ( )) → cierto
COLA_VACIA (ENCOLAR (queue, x)) → falso
DESENCOLAR (CREAR_COLA ( )) → error
PRIMERO_COLA (CREAR_COLA ( )) → error

$$\text{PRIMERO\_COLA (ENCOLAR (queue, x))} \rightarrow \begin{cases} x \text{ si } \text{COLA\_VACIA(queue)} = \text{true} \\ \text{PRIMERO\_COLA (queue) sino} \end{cases}$$

Estos axiomas vienen a decir básicamente lo siguiente:

Una cola recien creada (*CREAR_COLA*) está vacía, mientras que una cola en la que, al menos, hemos puesto un elemento no está vacía.

Tanto intentar eliminar un elemento, como consultar el primer elemento de una cola recien creada, produce un error.

Y finalmente la consulta de una cola en la que hemos insertado un nuevo elemento 'x', devolverá 'x' si la cola estaba vacia, o el primero de la cola si la cola ya contenia otros elementos.

12.2. Representación de las Colas en C++

Al igual que con las pilas, el primer paso de la implementación será decidir el prototipo de los métodos que vamos a utilizar.

La idea básica será seguir en la misma linea del tema anterior, buscando la máxima similitud entre las operaciones de pilas y colas.

Así las operaciones del interfaz que tendremos serán las siguientes, recordando que las operaciones que puedan devolver errores, devolverán un *boolean*.

```
class Cola
{
   public:
      Cola (void);
      bool Encolar (Valor);
      bool Desencolar (void);
      bool PrimeroCola (Valor &);
      bool ColaVacia (void);

   private:
      .??.
};
```

Para determinar concretamente cada una de estas operaciones y su implementación, es necesario especificar un tipo de representación para las colas. Dependiendo de esta representación tendremos diferentes implementaciones.

Ejemplo de utilización de colas

Con la interfaz propuesta ya somos capaces de utilizar la clase cola, y podemos realiza un ejemplo.

Realizar una función en C++ que nos diga el número de elementos que contiene una cola pasada como parámetro.

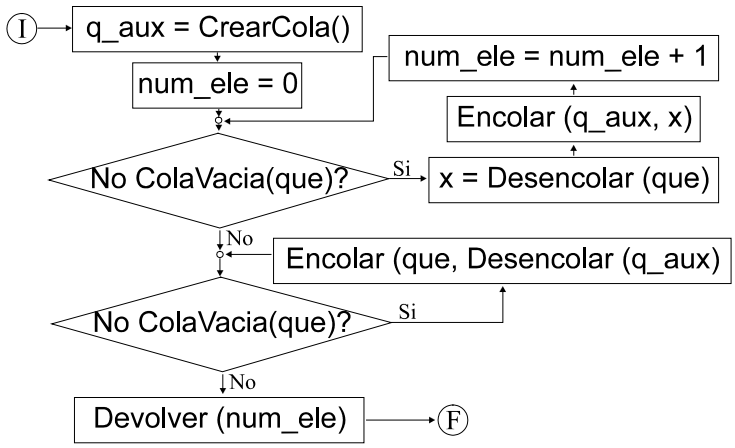
Para saber el número de elementos tendremos que ir desencolandolos de la cola y contando los elementos hasta que la cola se quede vacía. Para asegurarnos que la cola no queda modificada podemos pasarla por valor, o pasarla por referencia y si en algún momento la modificamos, devolverla antes de terminar la función a su valor original.

El algoritmo en este segundo caso sería:

Algoritmo ContarElementos
Entradas: Cola que;
Salidas: Cola que;
Entero num_ele;
Variables Cola q_aux;
Entero x;
Inicio
q_aux = CrearCola ()
num_ele = 0
Mientras (No ColaVacia(que)) **Hacer**
x = Desencolar (que)
Encolar (q_aux, x)
num_ele = num_ele + 1
Fin_mientras

Mientras (No ColaVacia (q_aux)) **Hacer**
Encolar (que, Desencolar (q_aux))
Fin_mientras
Devolver (num_ele)
Fin

El organigrama correspondiente a este algoritmo sería el siguiente:



Si aplicamos la interfaz que hemos definido para la clase cola en C++, podemos realizar la función en C++:

```
int ContarElementos (Cola & que)
{
   int num_ele;
   Cola q_aux;
   Valor x;

   num_ele = 0;

   while (!que.ColaVacia () )
   {
      b_aux = que.Desencolar (x);
      b_aux = q_aux.Encolar (x);
      num_ele++;
   }

   while (!q_aux.ColaVacia () )
   {
      b_aux = q_aux.Desencolar (x);
      b_aux = que.Encolar (x);
   }

   return num_ele;
}
```

No tenemos en cuenta el posible error del método **`Desencolar`***, porque para llegar a esa llamada hemos tenido en cuenta que la cola no esté vacía. El error de* **`Encolar`** *tampoco lo hemos tenido en cuenta porque si los elementos cabían en la cola original, también van a caber en la cola auxiliar.*

Aunque podríamos controlar la cola vacía a través del error que devuelve la función desencolar:

```
int ContarElementos (Cola & que)
{
   int num_ele;
   Cola q_aux;
   Valor x;

   num_ele = 0;

   while (!que.Desencolar (x) )
```

```
        {
           b_aux = q_aux.Encolar (x);
           num_ele++;
        }

        while (!q_aux.Desencolar (x) )
           b_aux = que.Encolar (x);

        return num_ele;
     }
```

Implementación mediante estructuras estáticas: Colas lineales

La representación de una cola finita en forma de vector es una tarea algo más compleja que la realizada para el caso de las pilas. Además de un array unidimensional, son necesarias un par de variables que indiquen dónde está el inicio de la cola y dónde el final.

Si son **ini** y **fin** las dos variables que apuntan a los extremos de la estrutura, normalmente se adopta el convenio de que la variable **ini** sea siempre la posición real del primer elemento y que la variable **fin** siempre apunte a la siguiente posicion de la cola donde podemos insertar nueva información. De esta manera, se cumplirá que **ini=fin** si y sólo si la cola está vacía, y la condición inicial para indicar que se ha creado una cola vacía será **ini=fin=0**.

Tal como hicimos con la definición del tipo pila, la definición 'compacta' del tipo cola sería:

```
class Cola
{
   public:
      .??.

   private:
      Vector info;
      int ini, fin;
};
```

Con este esquema de representación, se puede pasar a especificar el conjunto de operaciones necesarias para definir una cola:

Operación CREAR_COLA

Esta operación consistirá en definir la variable del tipo nuevo declarado, Cola (*array* que permitirá almacenar la información y las varibles que apuntarán a los extremos de la estructura) e iniciar los valores de manera que se indique explícitamente que la cola, trás la creación, está vacía.

---

(1) Variables
queue: Pila;

(2) Asignación de pila vacía
queue.ini ← 0
queue.fin ← 0

Esta operación, en C++, se convierte en el constructor por defecto:

```
Cola::Cola (void)
{
   ini = 0;
   fin = 0;
}
```

Operación COLA_VACIA

Con las condiciones establecidas, basta comprobar si los valores del inicio y el final de la cola son iguales:

```
Algoritmo Cola_Vacia
Entrada
        queue: Cola
Salida
        (CIERTO, FALSO)
Inicio
        Si ( queue.ini = queue.fin ) entonces
                Devolver ( CIERTO )
        Sino
                Devolver ( FALSO )
        Fin_si
Fin
```

```
bool Cola::ColaVacia (void)
{
   bool b_aux;

   if (ini == fin)
      b_aux = true;
   else
      b_aux = false;

   return b_aux;
}
```

O lo que sería lo mismo, devolver directamente la comparación **`ini == fin`**.

```
bool Cola::ColaVacia (void)
{
   return ini == fin;
}
```

Operación de inserción de información (ENCOLAR)

Al igual que en las pilas, la inserción de elementos está condicionada por la representación que se hace de la estructura. Al representar la cola con un array (tamaño finito), es preciso comprobar previamente si existe espacio disponible en la cola para almacenar más información. En el caso en que se pueda insertar la nueva información, la colocaremos en el lugar correspondiente dentro del vector y actualizaremos las marcas de inicio y final de la cola.

```
Algoritmo Encolar
Entradas
        x: Valor                          {* elemento que desea insertar *}
        queue: Cola de Valor
Salidas
        queue
Inicio
```

```
        {* comprobar si en la cola se pueden insertar más elementos        *}
        {* esto es necesario por el tipo de representación de la estructura *}
        Si ( queue.fin = MAX ) entonces
                Error "cola llena"
        Sino
                queue.info [queue.fin] ← x
                queue.fin ← queue.fin + 1
        Fin_sino
    Fin
```

```
bool Cola::Encolar (Valor x)
{
   bool error;

   if (fin == MAX)
      error = true;
   else
   {
      error = false;
      info [fin] = x
      fin++;
   }

   return error;
}
```

Operación de consulta de información (PRIMERO_COLA)

La consulta se hace a través de la posición que indica el campo **`Ini`**, y sólo si la cola no está vacía.

```
    Algoritmo Primero_Cola
    Entradas
        queue: Cola de Valor
    Salidas
        Valor
    Inicio
        {* comprobar si existe información en la cola                  *}
        {* esta operación no depende de la representación, siempre es necesaria *}
        Si ( Cola_Vacia ( queue ) ) entonces
                Error "cola vacia"
        sino
                Devolver ( queue.info [queue.ini] )
        Fin_si
    Fin
```

```
bool Cola::PrimeroCola (Valor & x)
{
   bool error;

   if (ColaVacia () )
      error = true;
   else
   {
      error = false;
      x = info[ini];
   }
   return error;
}
```

Operación de eliminación de información (DESENCOLAR)

Para eliminar elementos de la estructura, es preciso determinar si realmente hay información que extraer, sino el algoritmo debe detectar el error, independientemente de la representación concreta de la cola.

```
Algoritmo Desencolar
Entradas
        queue: Cola de Valor
Salidas
        queue: Cola, x: Valor
Inicio
        {* comprobar si se pueden eliminar elementos de la cola                *}
        {* esta operación no depende de la representación, siempre es necesaria *}
        Si ( Cola_Vacia (queue) ) entonces
                Error "cola vacia"
        sino
                {* Esta operación no sería realmente necesaria *}
                queue.ini ← queue.ini + 1
        Fin_si
Fin
```

```
bool Cola::Desencolar (void)
{
   bool error;

   if (ColaVacia () )
      error = true;
   else
   {
      error = false;
      ini++;
   }
   return error;
}
```

Implementación mediante estructuras estáticas: Colas circulares

Hay que tener en cuenta que, de hecho, la condición de cola llena (**`queue.fin = MAX`**), considerada en la operación de inserción, no indica necesariamente que existan n elementos en la cola, ya que es posible que exista espacio libre, por haber ido borrando elementos en las primeras posiciones del *array*.

Una solución obvia a este problema podría ser desplazar todos los elementos hacia la izquierda, cada vez que se produce una operación de borrado, hasta alcanzar el principio del *array*. Sin embargo, ésto no resulta demasiado eficiente, sobre todo cuando existen muchos elementos en la cola y la operaciones de borrado e inserción son muy frecuentes.

Por lo tanto, la representación de una cola especificada por las anteriores operaciones, puede dar lugar, en general, a una utilización ineficiente del espacio reservado para la estructura.

Una representación más eficiente se obtiene viendo el array donde guardamos la información contenida en la cola como si fuese circular. De esa manera, cuando se de el caso **`queue.Fin=MAX`**, será posible insertar nuevos elementos en la cola si los primeros elementos del *array* están libres.

Para trabajar de forma sencilla con esta representación será conveniente definir una operación auxiliar que nos lleve de un indice a su siguiente dentro de la nueva secuencia circular de índices. Esta operación se limitará a incrementar el índice si éste es menor que MAX y a volver a empezar en cero si se alcanza el valor de MAX.

Algoritmo Siguiente
Entradas
ind: 0..MAX - 1
Salidas
0..MAX - 1
Inicio
{ La operación* **mod** *devuelve el valor del resto de la division entera entre ind y MAX *}*
Devolver ((ind + 1) **mod** MAX)
Fin

El método **`siguiente`** quedará, en la declaración del interfaz de la clase, como un método privado, y su codificación en C++ será como sigue:

```
int Cola::Siguiente (int ind)
{
   return (ind + 1) % MAX;
}
```

Con esta nueva representación, la comprobación de si una cola está o no vacía se mantiene (**`queue.ini = queue.fin`**). Sin embargo, el resto de operaciones cambia ligeramente, ya que los elementos ya no se insertarán o consultarán en la posición incrementada del índice, sino en la posición "siguiente" siguiendo el patrón circular.

En primer lugar la iniciación de la estrucutra puede seguir haciéndose asignando los índices de inicio y fin de cola a cero.

Apoyándonos en la función auxiliar siguiente, los algoritmo de inserción, consulta y borrado quedarían de la siguiente forma:

```
Algoritmo Encolar
Entradas
        x: Valor                                    {* elemento que se desea insertar *}
        queue: Cola de Valor
Salidas
        queue
Inicio
        {* comprobar si en la cola se pueden insertar más elementos          *}
        {* esto es necesario por el tipo de representación de la estructura  *}
        Si ( Siguiente ( queue.fin ) = queue.ini ) entonces
                Error "cola llena"
        Sino
                queue.info [queue.fin] ← x
                queue.fin ← Siguiente (queue.fin)
        Fin_sino
Fin
```

Destacar en este algoritmo la siguiente consideración: Si consideraramos la posibilidad de insertar elementos en el vector hasta ocupar todos los elementos posibles del *array* tendríamos que la condición de cola llena y cola vacia sería la misma (**`queue.ini = queue.fin`**). Por ello, habitualmente se mantiene un elemento vacío entre la última posición ocupada y la última, de manera que la consición de cola vacía se mantiene, mientras que la de cola llena se transforma en: **`Siguiente (queue.fin) = queue.ini`**.

Si deseasemos utilizar ese hueco, de manera que la condición de cola llena y cola vacía no pudiese distinguirse, sería necesario añadir una variable lógica que nos indicase el estado real de la cola (si vacia o llena), sabiendo que sólo podemos llenar una cola tras hacer inserciones o sólo podemos dejarla vacía tras eliminar elementos.

El algoritmo propuesto traducido a C++ quedará:

```
bool Cola::Encolar (Valor x)
{
   bool error;

   if (Siguiente (fin) == ini)
      error = true;
   else
   {
      error = false;

      info[fin] = x;
      fin = Siguiente (fin);
   }
   return error;
}
```

El método **`PrimeroCola`** no cambia de ninguna manera, y el método **`Desencolar`** sólo cambia en el avance del índice '**`ini`**'

```
Algoritmo Desencolar
Entradas
        queue: Cola de Valor
Salidas
        queue
Inicio
        {* comprobar si se pueden eliminar elementos de la cola                *}
        {* esta operación no depende de la representación, siempre es necesaria *}
        Si ( Cola_Vacia (queue) ) entonces
                Error "cola vacia"
        sino
                {* Esta operación no sería realmente necesaria *}
                queue.Ini ← Siguiente (queue.Ini)
        Fin_si
Fin
```

```
bool Cola::Desencolar (void)
{
   bool error;

   if (ColaVacia () )
      error = true;
   else
   {
      error = false;
      ini = Siguiente (ini);
   }
   return error;
}
```

Implementación mediante estructuras dinámicas: Colas enlazadas

Creación de una cola: Constructor por defecto

Al igual que en el caso de pilas, desarrollaremos tres pasos para la creación de la cola dinámica:

(1) Definición de los tipos necesarios:

```
Tipo
  Nodo_cola = Registro
                Info: Valor
                Sig : Puntero_a_Nodo_cola
              Fin_reg

Tipo
  Cola = Registro
           Ini, Fin: Puntero_a_Nodo_cola
         Fin_reg
```

(2) Declaración de una variable de este nuevo tipo (cola):

```
Var
  queue: Cola
```

(3) Iniciación de la estructura como vacía:

```
Algoritmo Iniciar_Cola
Entradas
        queue: Cola de Valor
Salidas
        queue
Inicio
        queu.ini ← NULO
        queu.fin ← NULO
Fin
```

En C++, estos pasos se resumirían en la declaración de la variable:

```
Cola que;
```

La declaración de los tipos necesarios:

```
struct nodo
{
   Valor info;
   Puntero sig;
};

typedef nodo * Puntero;
```

La inclusión de la información necesaria en la parte privada de la clase:

```
class Cola
{
   public:
      .??.

   private:
      Puntero ini, fin;
};
```

Y la implementación del método constructor por defecto de la clase:

```
Cola::Cola (void)
{
   ini = NULL;
   fin = NULL;
}
```

Creación de una cola: Constructor de copia

Ya comentamos en Pilas la necesidad, en el caso de representación de información con estructuras dinámicas, del constructor de copia.

Cuando se pasa un parámetro por valor, se realiza una copia de la información contenida en la variable que se pasa por valor, pero solo de la información contenida en él.

Así, si la variable contiene punteros a diferentes espacios de memoria, el paso por valor, en principio realizará una copia de estos valores sin más, de manera que la memoria referenciada seguirá siendo la misma, y no tendremos una copia real de la información guardada.

Si queremos tener una copia de toda la información, deberemos incluir entre los métodos de la clase, el constructor de copia.

El contructor de copia es un método que es llamado automáticamente cada vez que se realiza un paso de parámetros por valor del objeto en concreto. Si existe el constructor de copia, éste es llamado, sino se realiza sólo la copia de la información contenida en el objeto.

El prototipo del constructor de copia es:

```
Nombre_de_la_clase (const Nombre_de_la_clase &)
```

Al igual que el constructor por defecto, no devuelve ningún valor, ni siquiera **void**. Tiene como nombre, el nombre de la clase, y como único parámetro un objeto por referencia constante, que será el objeto del que queremos realizar una copia.

En el caso que tenemos, colas, el constructor de copia podría ser:

```
Cola::Cola (const Cola & que)
{
   Puntero p_aux, q_aux;

   ini = NULL;
   p_aux = que.ini;
   while (p_aux != NULL)
   {
      q_aux = new Nodo;
      q_aux->info = p_aux->info;
      if (ini == NULL)
         ini = q_aux;
      else
         fin->sig = q_aux;
      fin = q_aux;

      p_aux = p_aux->sig;
   }
}
```

Comprobación de cola vacía

De nuevo, la estructura estará vacía si y sólo si inicio no apunta a ningún nodo de la cola (es decir, apunta a NULO.)

```
Algoritmo Cola_Vacia
Entrada
        queue: Cola
Salida
        (CIERTO, FALSO)
Inicio
        Si (queue.ini = NULO) entonces
                Devolver (CIERTO)
        Sino
                Devolver (FALSO)
        Fin_si
Fin
```

```
bool Cola::ColaVacia (void)
{
   return ini == NULL;
}
```

Acceso al primer elemento de la cola

Se puede acceder a este elemento de la cola mediante el puntero inicio que lo referencia.

```
Algoritmo Primero_Cola
Entradas
        queue: Cola de Valor
Salidas
        Valor
Inicio
        {* comprobar si existe información en la cola                    *}
        {* esta operación no depende de la representación, siempre es necesaria  *}
        Si ( Cola_Vacia ( queue ) ) entonces
                Error "cola vacia"
        sino
                Devolver ( queue.info [queue.ini] )
        Fin_si
Fin
```

```
bool Cola::PrimeroCola (Valor & x)
{
   bool error;

   if (ColaVacia () )
      error = true;
   else
   {
      error = false;
      x = ini->info;
   }
   return error;
}
```

Operación de inserción

Al igual que en cualquier otra estructura dinámica, la inserción consta básicamente de tres pasos: El primero de reserva del espacio necesario para el nuevo elemento; el segundo de asignación del valor a insertar; y finalmente, el tercero de enlace del nuevo elemento en la estructura dinámica.

```
Algoritmo Encolar
Entrada
        queue: Cola de Valor
        x: Valor
Salida
        queue
Variable
        p_aux: puntero a Nodo_cola
Inicio
        p_aux ← Crear_Espacio
        p_aux^.Info ← x
        p_aux^.Sig ← NULO
        Si ( Cola_Vacia ( queue ) entonces
                queue.Ini ← p_aux
        Sino
                queue.Fin^.Sig ← p_aux
        Fin_si
        queue.Fin ← p_aux
fin
```

La traducción a C++ es inmediata:

```
bool Cola::Encolar (Valor x)
{
   bool error;
   Puntero p_aux;

   error = false;

   p_aux = new Nodo;
   p_aux->info = x;
   p_aux->sig = NULL;

   if (ColaVacia () )
      ini = p_aux;
   else
      fin->sig = p_aux;

   fin = p_aux;

   return error;
}
```

Operación de eliminación de un elemento

Como en cualquier estructura dinámica, la eleiminación de información se realiza básicamente en tres pasos: (1) Obtener la información del elemento a borrar; (2) desenlazar el elemento de la

estructura; y (3) finalmente liberar el espacio ocupado por el elemento para dejarlo accesible para futuras llamadas.

Algoritmo Desencolar
Entrada
queue: Cola de Valor
Salida
queue
x: Valor
Variable
p_aux: puntero a Nodo_Cola
Inicio
Si$_{(1)}$ (*Cola_Vacia* (queue) **entonces**
Error "*cola_vacia*"
Sino$_{(1)}$
{ 1 *}*
x ← queue.Ini^.Info
{ 2 *}*
p_aux ← queue.Ini
queue.Ini ← p_aux^.Sig
{ si tras borrar se vacia la cola, hay poner* Fin *a nulo *}*
Si$_{(2)}$ (*Cola_Vacia* (queue) **entonces**
queue.Fin ← **NULO**
Fin_si$_{(2)}$
{ 3 *}*
Liberar_Espacio (p-aux)
Fin_si$_{(1)}$
Fin

```
bool Cola::Desencolar (void)
{
   bool error;
   Puntero p_aux;

   if (ColaVacia () )
      error = true;
   else
   {
      error = false;

      p_aux = ini;
      ini = ini->sig;

      dispose p_aux;
   }
   return error;
}
```

Operación de borrado de la estructura: Destructor de la clase

Una variable, cuando acaba el bloque donde está declarada, termina su existencia y es liberado el espacio que ocupa en memoria. Es decir, el sistema marca como libre ese espacio y lo deja disponible para su reutilización.

En el caso de los objetos ocurre exactamente lo mismo, el espacio de datos que ocupa el objeto es liberado por el sistema.

En el caso en que toda la información guardada por el objeto este declarada dentro del objeto (estructuras estáticas) no existe ningún porblema: La memoria ocupada es liberada.

Las cosas son ligeramente diferentes en el caso en que parte de la información guardada por el objeto no esté totalmente en el interior del objeto, sino que el objeto contenga enlaces a la memoria ocupada por la información (caso dinámico). En este caso, la memoria que se libera cuando acaba el bloque donde está declarado el objeto tan solo es la contenida en el objeto, es decir, solo los enlaces que enlazan con la memoria donde está la información. Con esto, la memoria ocupada por la información queda 'ocupada', mientras que las referencias de acceso para acceder a ella desaparecen.

Esto es una cosa que debemos evitar, liberando antes de que desaparezca el ámbito de la variable, la memoria ocupada por la información.

Un algoritmo que libera toda la memoria ocupada por una cola dinámica podría ser:

```
Algoritmo LiberarCola
Entrada
        queue: Cola de Valor
Salida
        queue
Variable
        p_aux: puntero a Nodo_cola
Inicio
        Mientras (No Cola_Vacia (queue) hacer
                p_aux ← que.ini
                que.ini ← que.ini^.sig
                Liberar (p_aux)
        fin_mientras
fin
```

En C++, para facilitar este proceso existe el método 'destructor' de la clase.

Al igual que los constructores, que son llamadso automáticamente cada vez que es necesario, el destructor de la clase es llamado siempre que vaya a desaparecer el objeto. Si existe lo ejecuta y libera elespacio ocupado por el objeto, y si no existe se limita a liberar el espacio ocupado por el objeto.

Asi como puede haber varios constructores, sólo puede haber un destructor y su prototipo es:

```
~Nombre_De_La_Clase (void)
```

No devuleve ningún tipo, ni siquiera **`void`**, igual que el constructor por defecto. Tiene como nombre el símbolo '**`~`**' seguido del nombre de la clase. Y como parámetros sólo puede tener **`void`**.

```
Cola::~Cola (void)
{
   Puntero p_aux;

   while (!ColaVacia() )
   {
      p_aux = ini;
      ini = ini->sig;
      dispose (p_aux);
   }
}
```