Tema I: Estructuras Dinámicas

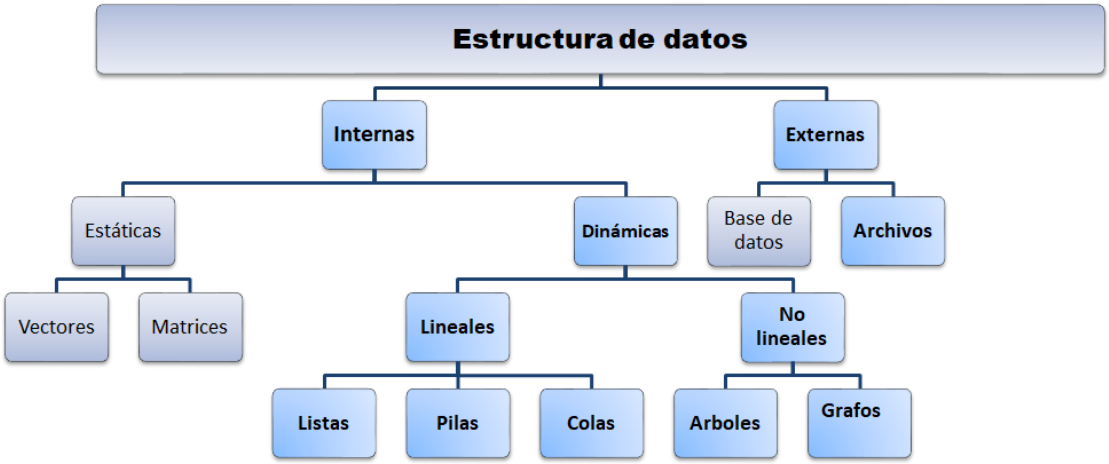
Estructuras dinámicas. Estructuras lineales. Listas enlazadas. Punteros. Declaración de listas enlazadas. Operaciones sobre listas enlazadas. Declaración de Listas circulares. Listas doblemente enlazadas. Pilas y Colas dinámicas.

Introducción

Las **estructuras de datos** son una forma de organizar los **datos** en la computadora, de tal manera que nos permita realizar unas operaciones con ellas de forma muy eficiente.

Las estructuras de datos pueden ser de dos tipos: Internas y externas. Además:
Estructura de datos estática: Una estructura de datos estática es aquella en la que el tamaño ocupado en memoria se define antes de que el programa se ejecute y no pueda modificarse dicho tamaño durante la ejecución del programa entre las estructuras de datos estáticas se encuentran los *array* (vectores y matrices).
Estructura de datos dinámica: Una estructura de datos dinámica es aquella en la que el tamaño ocupado en memoria puede modificarse durante la ejecución del programa.
Las variables que se crean y están disponibles durante la ejecución del programa se llaman variables continuas.
De esta manera se pueden adquirir posiciones adicionales de memoria a medida que se necesiten durante la ejecución del programa y liberarlas cuando no se necesiten.
Las estructuras dinámicas de datos se pueden dividir en dos grandes grupos: Lineales y no lineales.

Mapa conceptual



Estructuras lineales y no lineales

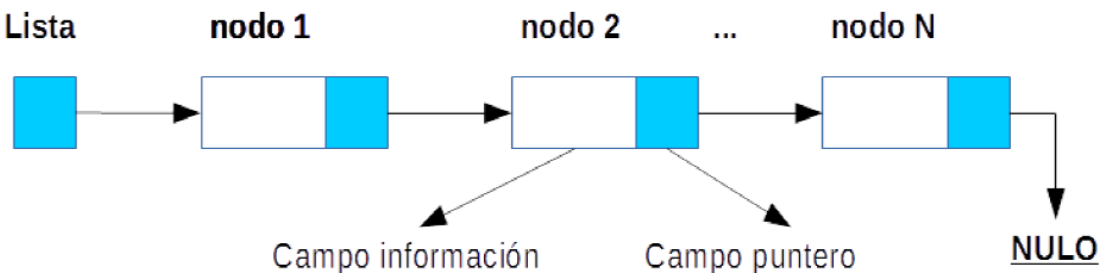
Las estructuras de datos simples se pueden combinar de varias maneras para formar estructuras más complejas. Las dos clases principales de estructuras de datos complejas son las lineales y las no lineales, dependiendo de la complejidad de las relaciones lógicas que representan. Las estructuras de datos lineales incluyen pilas, colas y listas. Las estructuras de datos no lineales incluyen grafos y árboles.
En este tema desarrollaremos estructuras de datos lineales: Listas, pilas y colas.

1.3 Listas enlazadas

La lista enlazada nos permite almacenar datos de una forma organizada, al igual que los vectores pero, a diferencia de estos, esta estructura es dinámica, por lo que no tenemos que saber "a priori" los elementos que puede contener.

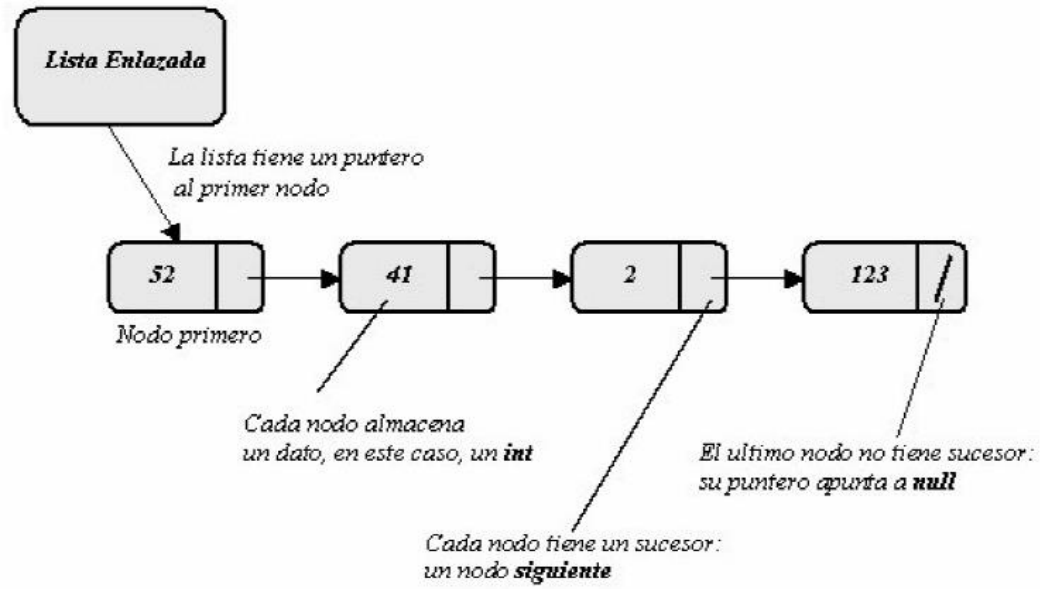
En una lista enlazada, cada elemento apunta al siguiente excepto el último que no tiene sucesor y el valor del enlace es `null`. Por ello los elementos son registros que contienen el dato a almacenar y un enlace al siguiente elemento. Los elementos de una lista, suelen recibir también el nombre de nodos de la lista.

Una lista enlazada es una colección lineal de elementos llamados nodos. El orden entre ellos se establece mediante punteros; direcciones o referencias a otros nodos.



Un nodo está constituido por dos partes:

- Un campo INFORMACIÓN: Que será del tipo de los datos que se quiera almacenar en la lista.
- Un campo LIGA de tipo puntero, que se utiliza para establecer la liga o el enlace con otro nodo de la lista.



Un programa accede a una lista enlazada mediante un apuntador al primer nodo en la lista.

Y se accede a cada nodo subsiguiente a través del miembro apuntador de enlace almacenado en el nodo anterior. El apuntador de enlace en el último nodo de una lista se establece en el valor nulo (0) para marcar el final de la lista.

Los datos se almacenan en forma dinámica en una lista enlazada; se crea cada nodo según sea necesario. Un nodo puede contener datos de cualquier tipo, incluyendo objetos de otras clases.

Tipos de listas ligadas

Listas **simplemente enlazadas**
Listas **doblemente enlazadas**. Colección de nodos en los que cada uno tiene dos apuntadores, uno a su predecesor y otro a su sucesor.
Listas circulares. El último nodo de la lista apunta al primero.

1.3.1 Punteros

Un puntero es una **variable que contiene la dirección de memoria** de otra variable. Los punteros permiten código más compacto y eficiente; utilizándolos en forma ordenada dan gran flexibilidad a la programación.

No hay que confundir una dirección de memoria con el contenido de esa dirección de memoria.

Dado un entero X = 125

Dirección

1408	1409	1412	1420	1421
…	….	125	…	…

La **dirección** de la variable X es 1412 y el **contenido** de la variable X es 125

Punteros

Entonces los punteros tienen dos “elementos”.

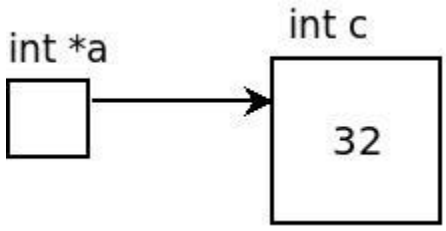
- Dirección de memoria en la que se almacena el valor
- El valor en sí

En c++ podemos trabajar con ambos elementos por separado declarando la variable como puntero. ¿Cómo declaramos un puntero?

```
int* a;
```

En este caso hemos declarado una variable a tipo puntero a entero, ¿qué quiere decir esto? que a apuntará a un entero. Como no le hemos dado ningún valor, inicialmente no está apuntando a nada. Observar siguiente código.

```
int c = 32;
int* a = &c;
```



Ahora el puntero a toma como valor la dirección de c. Si hicieramos

```
int c = 32;
int* a = &c;
cout << a;
```

Contrariamente a lo que se podría esperar, no se imprimiría por pantalla 32, porque la **variable** `a` **es una dirección de memoria**. Entonces, ¿cómo accedemos al valor al que apunta a? A través del operador dirección.

```
int c = 32;
int* a = &c;
cout << *a; //Muestra por pantalla 32.
```

Pero, ¿cuál es la gracia de los punteros?, veamos el siguiente ejemplo de código:

```
int c = 32;
int* a = &c;
cout << *a; // Muestra por pantalla 32.
c = 25;
cout << *a; // Muestra por pantalla 25.
*a = 16;
cout << c; // Muestra por pantalla 16.
```

Del mismo modo, a un puntero le podríamos asignar otro puntero.

```
int c = 32;
int* a = &c;
int* b = a; //a y b apuntan al valor de c

cout << *a; // Muestra por pantalla 32.
c = 25;
cout << *a; // Muestra por pantalla 25.
*a = 16;
cout << c; // Muestra por pantalla 16.
*b = 22;
```

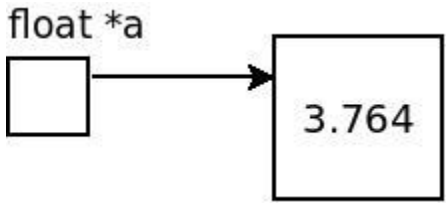
```
cout << c; // Muestra por pantalla 22.
cout << *a; // Muestra por pantalla 22.
cout << *b; // Muestra por pantalla 22.
```

Inicializando un puntero

En el ejemplo anterior hemos visto que a un puntero le podemos asignar la dirección de otra variable del mismo tipo. Pero también podemos crear un espacio *nuevo de memoria*.

```
float *a = new float{3.764};
  cout << *a ; // muestra por pantalla 3.764
```

En este ejemplo creamos un puntero a, que apuntará a un número decimal, y además reservamos espacio de memoria para albergar ese número decimal. Inicializamos el puntero con el valor 3.764.

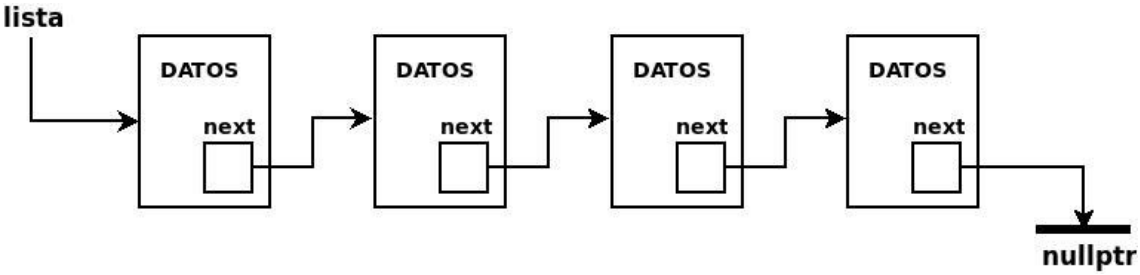


1.4 Declaración de listas enlazadas.

Para crear una lista enlazada recordamos los conocimientos sobre estructuras y asignación dinámica de memoria.

Crear una sencilla agenda que contiene el nombre y el número de teléfono.

Una lista enlazada simple necesita una estructura con varios campos, los campos que contienen los datos necesarios (nombre y teléfono) y otro campo que contiene un puntero a la propia estructura. Este puntero se usa para saber dónde está el siguiente elemento de la lista, para saber la posición en memoria del siguiente elemento.




```
struct _agenda {
        char nombre[20];
        char telefono[12];
      struct _agenda *siguiente;
                };
```

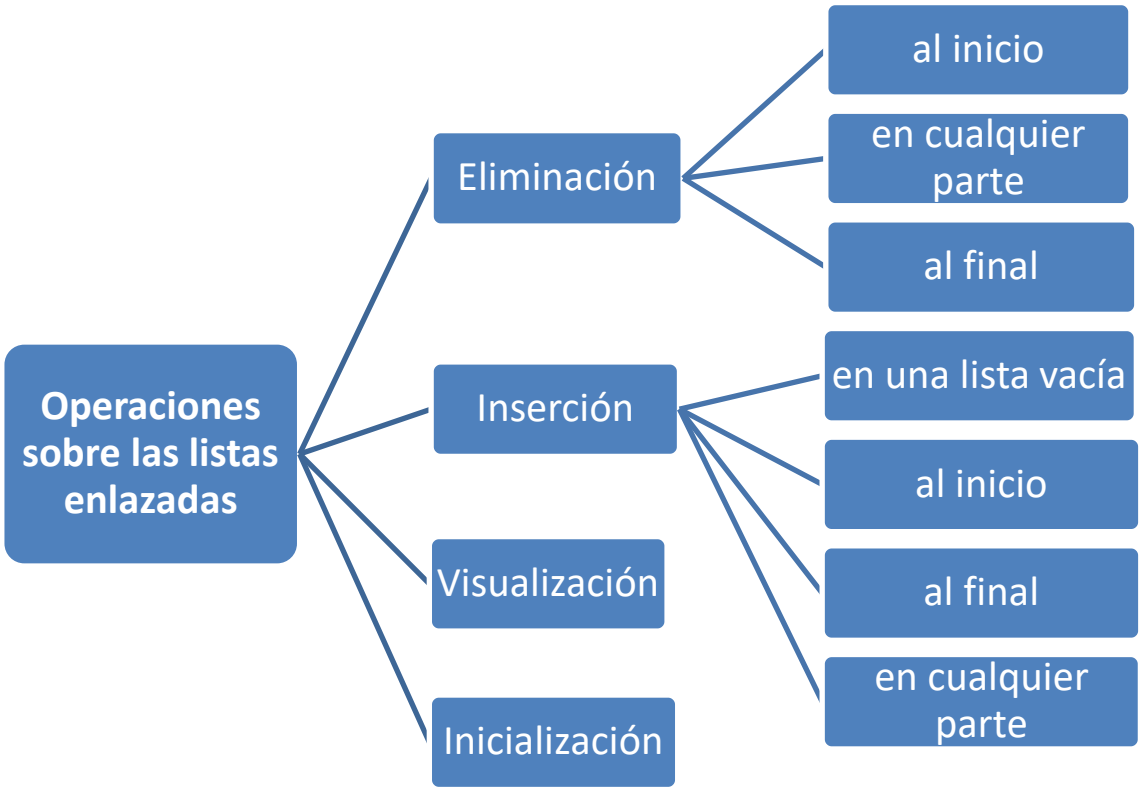
Esencialmente, una lista será representada como un puntero que señala al principio (o cabeza) de la lista.

```
typedef struct ElementoLista {
   char *dato;
   struct ElementoLista *siguiente;
} Elemento;
```

Para tener el control de la lista es preferible guardar determinados elementos: el primer elemento, el último elemento, el número de elementos. Para ello será empleado otra estructura (no es obligatorio, pueden ser utilizadas variables).

```
typedef struct ListaIdentificar {
     Elemento *inicio;
     Elemento *fin;
     int tamaño;
}Lista;
```

1.5 Operaciones sobre listas enlazadas.



1.5.1 Inicialización

```
void inicializacion (Lista *lista);
```

Esta operación debe ser hecha antes de otra operación sobre la lista. Esta comienza el puntero **inicio** y el puntero **fin** con el puntero NULL, y el **tamaño** con el valor 0.

```
void inicializacion (Lista *lista){
     lista->inicio = NULL;
     lista->fin = NULL;
tamaño = 0;
}
```

1.5.2 Inserción

A continuación el algoritmo de inserción y el registro de los elementos, pasos:

1. declaración del elemento que se va a insertar;
2. asignación de la memoria para el nuevo elemento;
3. llena el contenido del campo de datos;
4. actualización de los punteros hacia el primer y último elemento si es necesario.

Caso particular: en una lista con un único elemento, el primero es al mismo tiempo el último. Actualizar el tamaño de la lista

Casos para insertar elementos a una lista enlazada:

a) en una lista vacía,
b) al inicio de la lista,
c) al final de la lista,
d) en otra parte de la lista

a) Inserción en una lista vacía

```
int ins_en_lista_vacia (Lista *lista, char *dato);
```

La función retorna 1 en caso de error, si no devuelve 0.

Las etapas son asignar memoria para el nuevo elemento, completa el campo de datos de ese nuevo elemento, el puntero **siguiente** de este nuevo elemento apuntará hacia NULL (ya que la inserción es realizada en una lista vacía, se utiliza la dirección del puntero **inicio** que vale NULL), los punteros **inicio** y **fin** apuntaran hacia el nuevo elemento y el tamaño es actualizado.

```
/* Inserción en una lista vacia */

int  ins_en_lista_vacia (Lista * lista, char *dato){
Element  *nuevo_elemento;
if ((nuevo_elemento = (Element *) malloc (sizeof
(Element))) == NULL)
return -1;
if ((nuevo _elemento->dato = (char *) malloc (50 *
sizeof (char)))
== NULL)
return -1;

strcpy (nuevo_elemento->dato, dato);

nuevo_elemento->siguiente = NULL;
lista->inicio = nuevo_elemento;
lista->fin = nuevo_elemento;
lista->tamaño++;
return 0;
}
```

b) Inserción al inicio de la lista

```
int ins_inicio_lista (Lista *lista,char *dato);
```

Etapas: asignar memoria al nuevo elemento, rellenar el campo de datos de este nuevo elemento, el puntero **siguiente** del nuevo elemento apunta hacia el primer elemento, el puntero **inicio** apunta al nuevo elemento, el puntero **fin** no cambia, el tamaño es incrementado.

```
/* inserción al inicio de la lista */
int ins_inicio_lista (Lista * lista, char *dato){
  Element *nuevo_elemento;
  if ((nuevo_elemento = (Element *) malloc (sizeof
(Element))) == NULL)
    return -1;
  if ((nuevo_elemento->dato = (char *) malloc (50 *
sizeof (char)))
      == NULL)
    return -1;
  strcpy (nuevo_elemento->dato, dato);

  nuevo_elemento->siguiente = lista->inicio
  lista->inicio = nuevo_elemento;
  lista->tamaño++;
  return 0;
}
```

c) Inserción al final de la lista,

```
int ins_fin_lista (Lista *lista, Element *actual, char *dato);
```

Etapas: proporcionar memoria al nuevo elemento, rellenar el campo de datos del nuevo elemento, el puntero **siguiente** del último elemento apunta hacia el nuevo elemento, el puntero **fin** apunta al nuevo elemento, el puntero **inicio** no varía, el tamaño es incrementado:

```
/*inserción al final de la lista */
int ins_fin_lista (Lista * lista, Element * actual,
char *dato){
  Element *nuevo_elemento;
  if ((nuevo_elemento = (Element *) malloc (sizeof
(Element))) == NULL)
    return -1;
  if ((nuevo_elemento->dato = (char *) malloc (50 *
sizeof (char)))
      == NULL)
    return -1;
  strcpy (nuevo_elemento->dato, dato);

  actual->siguiente = nuevo_elemento;
  nuevo_elemento->siguiente = NULL;

  lista->fin = nuevo_elemento;

  lista->tamaño++;
  return 0;
}
```

d) Inserción en otra parte de la lista

```
int ins_lista (Lista *lista, char *dato,int pos);
```

La función arroja **-1** en caso de error, si no da **0**.

La inserción se efectuará después de haber pasado a la función una posición como argumento. Si la posición indicada no tiene que ser el último elemento. En ese caso se debe utilizar la función de inserción al final de la lista.

Etapas: asignación de una cantidad de memoria al nuevo elemento, rellenar el campo de datos del nuevo elemento, escoger una posición en la lista (la inserción se hará luego de haber elegido la posición), el puntero **siguiente** del nuevo elemento apunta hacia la dirección a la que apunta el puntero **siguiente** del elemento actual, el puntero **siguiente** del elemento actual apunta al nuevo elemento, los punteros **inicio** y **fin** no cambian, el tamaño se incrementa en una unidad:

```
/* inserción en otra parte de la lista */
int ins_lista (Lista * lista, char *dato, int pos){
  if (lista->tamaño < 2)
    return -1;
  if (pos < 1 || pos >= lista->tamaño)
    return -1;

  Element *actual;
  Element *nuevo_elemento;
  int i;

  if ((nuevo_elemento = (Element *) malloc (sizeof
(Element))) == NULL)
    return -1;
  if ((nuevo_elemento->dato = (char *) malloc (50 *
sizeof (char)))
      == NULL)
    return -1;

  actual = lista->inicio;
  for (i = 1; i < pos; ++i)
    actual = actual->siguiente;
  if (actual->siguiente == NULL)
    return -1;
  strcpy (nuevo_elemento->dato, dato);

  nuevo_elemento->siguiente = actual->siguiente;
  actual->siguiente = nuevo_elemento;
  lista->tamaño++;
  return 0;
```

1.5.3 Eliminación

A continuación un algoritmo para eliminar un elemento de la lista: uso de un puntero temporal para almacenar la dirección de los elementos a borrar, el elemento a eliminar se encuentra después del elemento actual, apuntar el puntero siguiente del elemento actual en dirección del puntero siguiente del elemento a eliminar, liberar la memoria ocupada por el elemento borrado, actualizar el tamaño de la lista.

Para eliminar un elemento de la lista hay varios casos:

a) Eliminación al inicio de la lista
b) Eliminación en otra parte de la lista.
c) Eliminación al final de la lista (penúltimo elemento)

a) Eliminación al inicio de la lista

```
int sup_inicio (Lista *lista);
```

La función devuelve **-1** en caso de equivocación, de lo contrario da **0**. Etapas: el puntero **sup_elem** contendrá la dirección del 1er elemento, el puntero **inicio** apuntara hacia el segundo elemento, el tamaño de la lista disminuirá un elemento:

```
/* Eliminación al inicio de la lista */
int sup_inicio (Lista * lista){
  if (lista->tamaño == 0)
    return -1;
  Element *sup_elemento;
  sup_element = lista->inicio;
  lista->inicio = lista->inicio->siguiente;
  if (lista->tamaño == 1)
    lista->fin = NULL;
  free (sup_elemento->dato);
  free (sup_elemento);
  lista->tamaño--;
  return 0;
}
```

b) Eliminación en otra parte de la lista.

```
int sup_en_lista (Lista *lista, int pos);
```

La función da **-1** en caso de error, si no devuelve **0**.

Etapas: el puntero **sup_elem** contendrá la dirección hacia la que apunta el puntero **siguiente** del elemento **actual**, el puntero **siguiente** del elemento actual apuntara hacia el elemento al que apunta el puntero **siguiente** del elemento que sigue al elemento **actual** en la lista. Si el elemento **actual** es el penúltimo elemento, el puntero **fin** debe ser actualizado. El tamaño de la lista será disminuido en un elemento:

```
/* eliminar un elemento en otro parte de la lista */
int sup_en_lista (Lista * lista, int pos){
  if (lista->tamaño <= 1 || pos < 1 || pos >= lista-
>tamaño)
    return -1;
  int i;
  Element *actual;
  Element *sup_elemento;
  actual = lista->inicio;

  for (i = 1; i < pos; ++i)
    actual = actual->siguiente;

  sup_elemento = actual->siguiente;
  actual->siguiente = actual->siguiente->siguiente;
  if(actual->siguiente == NULL)
          lista->fin = actual;
  free (sup_elemento->dato);
  free (sup_elemento);
  lista->tamaño--;
  return 0;
}
```

1.5.4 Visualización

Para mostrar la lista entera hay que posicionarse al inicio de la lista (el puntero inicio lo permitirá). Luego usando el puntero siguiente de cada elemento la lista es recorrida del primero al último elemento.

La condición para detener es proporcionada por el puntero siguiente del último elemento que vale NULL.

```
/* Visualización de la lista */
void visualización (Lista * lista){
  Element *actual;
  actual = lista->inicio;
  while (actual != NULL){
      printf ("%p - %s\n", actual, actual->dato);
      actual = actual->siguiente;
  }
}
```

1.6 Declaración de Listas circulares.

Una lista circular es una lista lineal en la que el último nodo a punta al primero.

Las listas circulares evitan excepciones en las operaciones que se realicen sobre ellas. No existen casos especiales, cada nodo siempre tiene uno anterior y uno siguiente.

En algunas listas circulares se añade un nodo especial de cabecera, de ese modo se evita la única excepción posible, la de que la lista esté vacía.

El nodo típico es el mismo que para construir listas abiertas:

```
struct nodo {
   int dato;
   struct nodo *siguiente;
};
```

Declaraciones de tipos para manejar listas circulares en C

Los tipos que definiremos normalmente para manejar listas cerradas son los mismos que para para manejar listas abiertas:

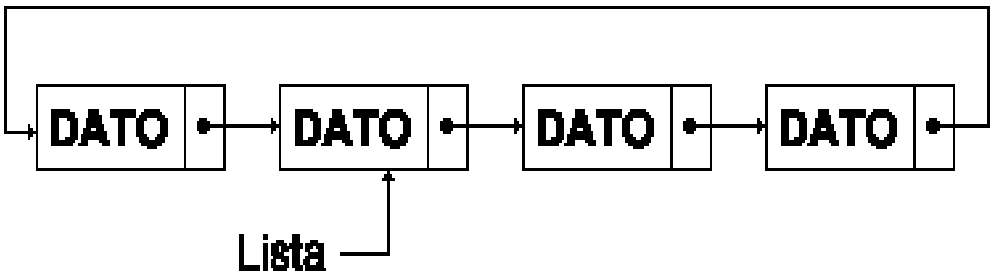
```
typedef struct _nodo {
   int dato;
   struct _nodo *siguiente;
} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Lista;
```

`tipoNodo` es el tipo para declarar nodos, evidentemente.

`pNodo` es el tipo para declarar punteros a un nodo.

Lista es el tipo para declarar listas, tanto abiertas como circulares. En el caso de las circulares, apuntará a un nodo cualquiera de la lista.

A pesar de que las listas circulares simplifiquen las operaciones sobre ellas, también introducen algunas complicaciones. Por ejemplo, en un proceso de búsqueda, no es tan sencillo dar por terminada la búsqueda cuando el elemento buscado no existe.



Por ese motivo se suele resaltar un nodo en particular, que no tiene por qué ser siempre el mismo. Cualquier nodo puede cumplir ese propósito, y puede variar durante la ejecución del programa.

Otra alternativa que se usa a menudo, y que simplifica en cierto modo el uso de listas circulares es crear un nodo especial de hará la función de nodo cabecera. De este modo, la lista nunca estará vacía, y se eliminan casi todos los casos especiales.

Ejemplo de lista circular en C

Construiremos una lista cerrada para almacenar números enteros. Haremos pruebas insertando varios valores, buscándolos y eliminándolos alternativamente para comprobar el resultado.

Algoritmo de la función "Insertar"

1. Si lista está vacía hacemos que lista apunte a nodo.
2. Si lista no está vacía, hacemos que nodo->siguiente apunte a lista->siguiente.
3. Después que lista->siguiente apunte a nodo.

```
void Insertar(Lista *lista, int v) {
   pNodo nodo;

   // Creamos un nodo para el nuevo valor a insertar
   nodo = (pNodo)malloc(sizeof(tipoNodo));
   nodo->valor = v;

   // Si la lista está vacía, la lista será el nuevo nodo
   // Si no lo está, insertamos el nuevo nodo a continuación
   // del apuntado por lista
   if(*lista == NULL) *lista = nodo;
   else nodo->siguiente = (*lista)->siguiente;
   // En cualquier caso, cerramos la lista circular
   (*lista)->siguiente = nodo;
}
```

1.7 Listas doblemente enlazadas.

Una lista doblemente enlazada es una lista lineal en la que cada nodo tiene dos enlaces, uno al nodo siguiente, y otro al anterior.

Las listas doblemente enlazadas no necesitan un nodo especial para acceder a ellas, pueden recorrerse en ambos sentidos a partir de cualquier nodo, esto es porque a partir de cualquier nodo, siempre es posible alcanzar cualquier nodo de la lista, hasta que se llega a uno de los extremos.

El nodo típico es el mismo que para construir las listas que hemos visto, salvo que tienen otro puntero al nodo anterior:

```
struct nodo {
   int dato;
   struct nodo *siguiente;
   struct nodo *anterior;
};
```

Declaraciones de tipos para manejar listas doblemente enlazadas en C

Para C, y basándonos en la declaración de nodo que hemos visto más arriba, trabajaremos con los siguientes tipos:

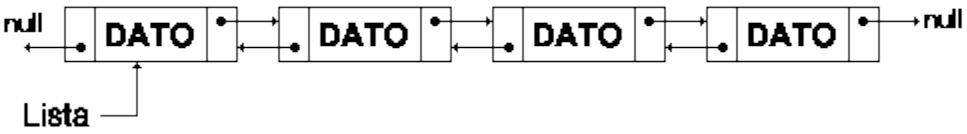
```
typedef struct _nodo {
   int dato;
   struct _nodo *siguiente;
   struct _nodo *anterior;
} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Lista;
```

`tipoNodo` es el tipo para declarar nodos, evidentemente.

`pNodo` es el tipo para declarar punteros a un nodo.

Lista es el tipo para declarar listas abiertas doblemente enlazadas. También es posible, y potencialmente útil, crear listas doblemente enlazadas y circulares.

El movimiento a través de listas doblemente enlazadas es más sencillo, y como veremos las operaciones de búsqueda, inserción y borrado, también tienen más ventajas.



Lista es el tipo para declarar listas abiertas doblemente enlazadas. También es posible, y potencialmente útil, crear listas doblemente enlazadas y circulares.

El movimiento a través de listas doblemente enlazadas es más sencillo, y como veremos las operaciones de búsqueda, inserción y borrado, también tienen más ventajas.

Ejemplo de lista doblemente enlazada en C

Como en el caso de los ejemplos anteriores, construiremos una lista doblemente enlazada para almacenar números enteros. Para aprovechar mejor las posibilidades de estas listas, haremos que la lista esté ordenada. Haremos pruebas insertando varios valores, buscándolos y eliminándolos alternativamente para comprobar el resultado.

Algoritmo de inserción

1) El primer paso es crear un nodo para el dato que vamos a insertar.
2) Si Lista está vacía, o el valor del primer elemento de la lista es mayor que el del nuevo, insertaremos el nuevo nodo en la primera posición de la lista.
3) En caso contrario, buscaremos el lugar adecuado para la inserción, tenemos un puntero "anterior". Lo inicializamos con el valor de Lista, y avanzaremos mientras anterior->siguiente no sea NULL y el dato que contiene anterior->siguiente sea menor o igual que el dato que queremos insertar.
4) Ahora ya tenemos anterior señalando al nodo adecuado, así que insertamos el nuevo nodo a continuación de él.

```
void Insertar(Lista *lista, int v) {
   pNodo nuevo, actual;

   /* Crear un nodo nuevo */
   nuevo = (pNodo)malloc(sizeof(tipoNodo));
   nuevo->valor = v;

   /* Colocamos actual en la primera posición de la lista */
   actual = *lista;
   if(actual) while(actual->anterior) actual = actual-
>anterior;

   /* Si la lista está vacía o el primer miembro es mayor que
el nuevo */
```

```
   if(!actual || actual->valor > v) {
      /* Añadimos la lista a continuación del nuevo nodo */
      nuevo->siguiente = actual;
      nuevo->anterior = NULL;
      if(actual) actual->anterior = nuevo;
      if(!*lista) *lista = nuevo;
   }
   else {
      /* Avanzamos hasta el último elemento o hasta que el
siguiente tenga
         un valor mayor que v */
      while(actual->siguiente && actual->siguiente->valor <=
v)
         actual = actual->siguiente;
      /* Insertamos el nuevo nodo después del nodo anterior
*/
      nuevo->siguiente = actual->siguiente;
      actual->siguiente = nuevo;
      nuevo->anterior = actual;
      if(nuevo->siguiente) nuevo->siguiente->anterior =
nuevo;
   }
}
```

1.8 Pilas y Colas dinámicas

Uno de los conceptos que más se emplean en las estructuras de datos, en la elaboración de programas, son las pilas. Éstas son aplicadas en cuanto a las restricciones sobre el acceso a los datos del arreglo, ya sea para insertar o eliminar elementos, actualizando el contenido de los registros.

Iniciemos por definir el término fundamental de una pila como "*una lista de elementos en la cual se puede insertar y eliminar elementos sólo por uno de los dos extremos*." (Osvaldo Cairó/Silvia Guardati*, Estructuras de datos*)

Para tener una comprensión más clara de lo que es una pila, imagine el acomodo de latas de un producto X en un centro comercial. O bien el apilamiento de libros en una biblioteca.

Según las imágenes anteriores, podemos deducir el razonamiento de cómo extraer un elemento de la pila, los componentes de una pila serán empleados en orden inverso al que se colocaron. Es decir, el último en entrar debe ser el primero en salir.

Las pilas son estructuras utilizadas muy a menudo como herramientas de programación que permiten el acceso solo a un elemento a la vez: el último elemento insertado. La mayoría de los procesadores utilizan una arquitectura basada en pilas.

La pila se considera un grupo ordenado de elementos, teniendo en cuenta que el orden de los mismos depende del tiempo que lleven “dentro” de la estructura. Las pilas son empleadas en el desarrollo de sistemas informáticos y software en general. Por ejemplo, el sistema de soporte en tiempo de compilación y ejecución, utiliza una pila para llevar la cuenta de los parámetros de procedimientos y funciones, variables locales, globales y dinámicas. Este tipo de estructuras también son utilizadas para traducir expresiones aritméticas o cuando se quiere recordar una secuencia de acciones u objetos en el orden inverso del ocurrido.

Ahora conoceremos el concepto de Cola y se definirá como “una estructura de almacenamiento donde los datos van a ser insertados por un extremo y serán extraídos por otro”. El concepto anterior se puede simplificar como, el primer elemento en entrar debe ser el primero en salir.

Los ejemplos sobre este mecanismo de acceso, son visibles en las filas de un banco, los clientes llegan (entran) se colocan en la fila y esperan su turno para salir.

Las pilas no son estructuras de datos fundamentales, esto es, no se encuentran definidas como tales en los lenguajes de programación. Las pilas pueden representarse mediante lo siguiente:

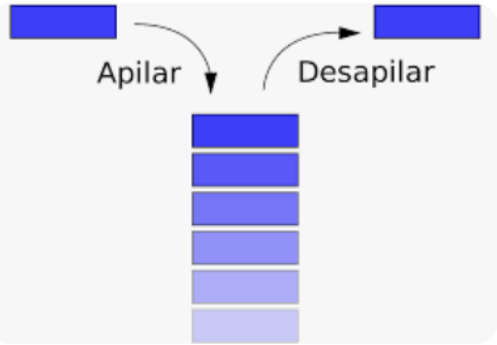
- Arreglos
- Listas enlazadas

1.8.1 Pilas

Una pila es una estructura dinámica que «apila» elementos de forma que para llegar al primero, hay que quitar todos los nodos que se hayan añadido después. Utiliza **LIFO** (*Last Input First Output*) que significa que el último que entra es el primero que saldrá, acrónimo que refleja la característica más importante de las pilas.

Una pila es un tipo de lista en el que todas las inserciones y eliminaciones de elementos se realizan por el mismo extremo de la lista.

Ya que la inserción es siempre hecha al inicio de la lista, el primer elemento de la lista será el último elemento ingresado, por lo tanto estará en la cabeza de la pila.



Una pila es un tipo especial de lista en la que sólo se pueden insertar y eliminar nodos en uno de los extremos de la lista. Estas operaciones se conocen como "push" y "pop", respectivamente "empujar" y "tirar". Además, las escrituras de datos siempre son inserciones de nodos, y las lecturas siempre eliminan el nodo leído.

1.8.1.1 Implementación.

Declaraciones de tipos para manejar pilas en C.

Para definir un elemento de la **pila** será utilizado el tipo **struct**. El elemento de la pila contendrá un campo **dato** y un puntero **siguiente**.

El puntero **siguiente** tiene que ser de la misma clase que el elemento, de lo contrario no va a poder apuntar hacia el elemento. El puntero **siguiente** permitirá el acceso al próximo elemento.

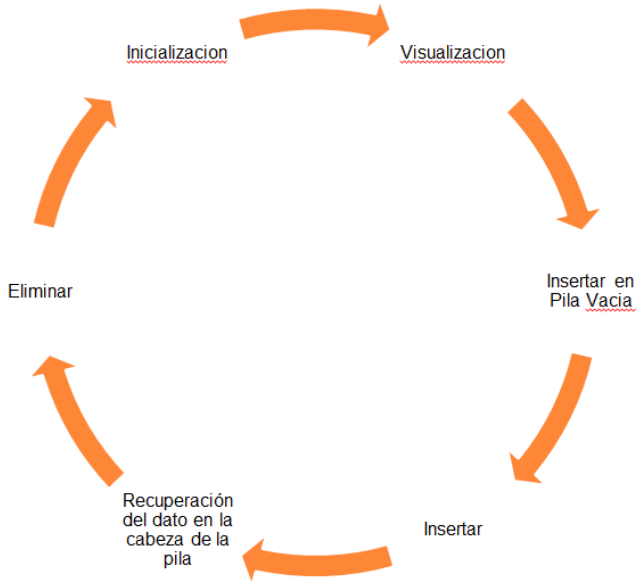
```
typedef struct _nodo {
   int dato;
   struct _nodo *siguiente;
} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Pila;
```

`tipoNodo` es el tipo para declarar nodos, evidentemente.

`pNodo` es el tipo para declarar punteros a un nodo.

`Pila` es el tipo para declarar pilas.

1.8.1.2 Operaciones básicas con pilas



a) Inicialización

Modelo de la función: `void inicialización (Pila *tas);`

Esta operación debe ser realizada antes de cualquier otra operación sobre la pila. Esta inicializa el puntero **inicio** con el valor **NULL** y el tamaño con el valor 0.

```
void inicialización (Pila * tas) {
          tas->inicio = NULL;
          tas->tamaño = 0;
     }
```

b) Visualización

Para mostrar la pila entera, es necesario posicionarse al inicio de la pila (el puntero **inicio** lo permitirá).

Luego, utilizando el puntero **siguiente** de cada elemento, la pila es recorrida del primero hacia el último elemento.

La condición para detenerse es determinada por el **tamaño** de la pila.

```
/* visualización de la pila */
void muestra (Pila * tas){
  Elemento *actual;
  int i;
  actual = tas->inicio;

  for(i=0;i<tas->tamaño;++i){
    printf("\t\t%s\n", actual->dato);
    actual = actual->siguiente;
  }
}
```

c) Insertar

El algoritmo de inserción y el registro de los elementos: declaración del elemento que va a insertarse, asignación de la memoria para el siguiente elemento, introducir el contenido del campo de los datos, actualizar el puntero **inicio** hacia el primer elemento (la cabeza de la pila). Actualizar el **tamaño** de la pila, tendrá tamaño **1** después de la primera inserción

Modelo de la función: `int apilar (Pila *tas, char *dato);`

La inserción siempre se hace en la parte superior de la pila.

```
/* apilar (añadir) un elemento en la pila */
int apilar (Pila * tas, char *dato){
  Elemento *nuevo_elemento;
  if ((nuevo_elemento = (Elemento *) malloc (sizeof (Elemento))) ==
NULL)
    return -1;
  if ((nuevo_elemento->dato = (char *) malloc (50 * sizeof (char)))
      == NULL)
    return -1;
  strcpy (nuevo_elemento->dato, dato);
  nuevo_elemento->siguiente = tas->inicio;
  tas->inicio = nuevo_elemento;
  tas->tamaño++;
}
```

d) Recuperación del dato en la cabeza de la pila

Para recuperar el dato en la cabeza de la pila sin eliminarlo, he utilizado una macro. La macro lee los datos en la parte superior de la pila utilizando el puntero **inicio**.

```
#define pila_dato(tas) tas->inicio->dato
```

e) Eliminar

Simplemente hay que eliminar el elemento hacia el cual apunta el puntero inicio. Esta operación no permite recuperar el dato en la cabeza de la pila, solo eliminarlo.

Modelo de la función: `int desapilar (Pila *tas);`

La función da como resultado -1 en caso de error, si no devuelve 0.

Las etapas:

El puntero **sup_elemento** contendrá la dirección del primer elemento.

El puntero **inicio** apuntará hacia el segundo elemento (después de la eliminación del primer elemento, el segundo elemento estará en la cabeza de la pila).

El **tamaño** de la pila disminuirá un elemento.

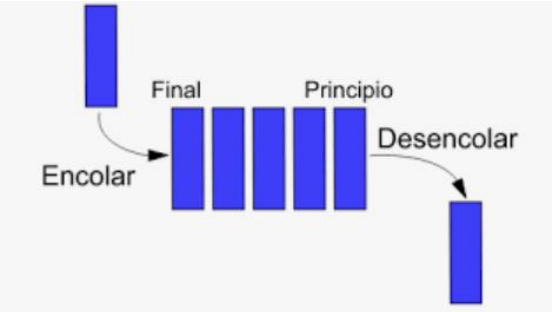
```
int desapilar (Pila * tas){
  Elemento *sup_elemento;
  if (tas->tamaño == 0)
    return -1;
  sup_elemento = tas->inicio;
  tas->inicio = tas->inicio->siguiente;
  free (sup_elemento->dato);
  free (sup_elemento);
  tas->tamaño--;
  return 0;
}
```

1.8.1.3 Ejemplo en C

```
#include <stdlib.h>
#include <stdio.h>

typedef struct _nodo {
   int valor;
   struct _nodo *siguiente;
} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Pila;

/* Funciones con pilas: */
void Push(Pila *l, int v);
int Pop(Pila *l);

int main()
{
   Pila pila = NULL;
   pNodo p;

   Push(&pila, 20);
   Push(&pila, 10);
   Push(&pila, 40);
   Push(&pila, 30);

   printf("%d, ", Pop(&pila));
   printf("%d, ", Pop(&pila));
   printf("%d, ", Pop(&pila));
   printf("%d\n", Pop(&pila));

   system("PAUSE");
   return 0;
}

void Push(Pila *pila, int v)
{
   pNodo nuevo;

   /* Crear un nodo nuevo */
   nuevo =
(pNodo)malloc(sizeof(tipoNodo)
);
   nuevo->valor = v;

   /* Añadimos la pila a
continuación del nuevo nodo */
   nuevo->siguiente = *pila;
   /* Ahora, el comienzo de
nuestra pila es en nuevo nodo
*/
   *pila = nuevo;
}

int Pop(Pila *pila)
{
   pNodo nodo; /* variable
auxiliar para manipular nodo
*/
   int v;      /* variable
auxiliar para retorno */

   /* Nodo apunta al primer
elemento de la pila */
   nodo = *pila;
   if(!nodo) return 0; /* Si
no hay nodos en la pila
retornamos 0 */
   /* Asignamos a pila toda la
pila menos el primer elemento
*/
   *pila = nodo->siguiente;
   /* Guardamos el valor de
retorno */
   v = nodo->valor;
   /* Borrar el nodo */
   free(nodo);
   return v;
}
```

1.8.2 Colas

Una cola es un tipo especial de lista abierta en la que sólo se puede insertar nodos en uno de los extremos de la lista y sólo se pueden eliminar nodos en el otro. Además, como sucede con las pilas, las escrituras de datos siempre son inserciones de nodos, y las lecturas siempre eliminan el nodo leído.



Este tipo de lista es conocido como lista FIFO (*First In First Out*), el primero en entrar es el primero en salir.

El símil cotidiano es una cola para comprar, por ejemplo, las entradas del cine. Los nuevos compradores sólo pueden colocarse al final de la cola, y sólo el primero de la cola puede comprar la entrada.

1.8.2.1 Implementación.

Declaraciones de tipos para manejar colas en C.

El nodo típico para construir cola es el mismo que vimos en los apartados anteriores para la construcción de listas y pilas:

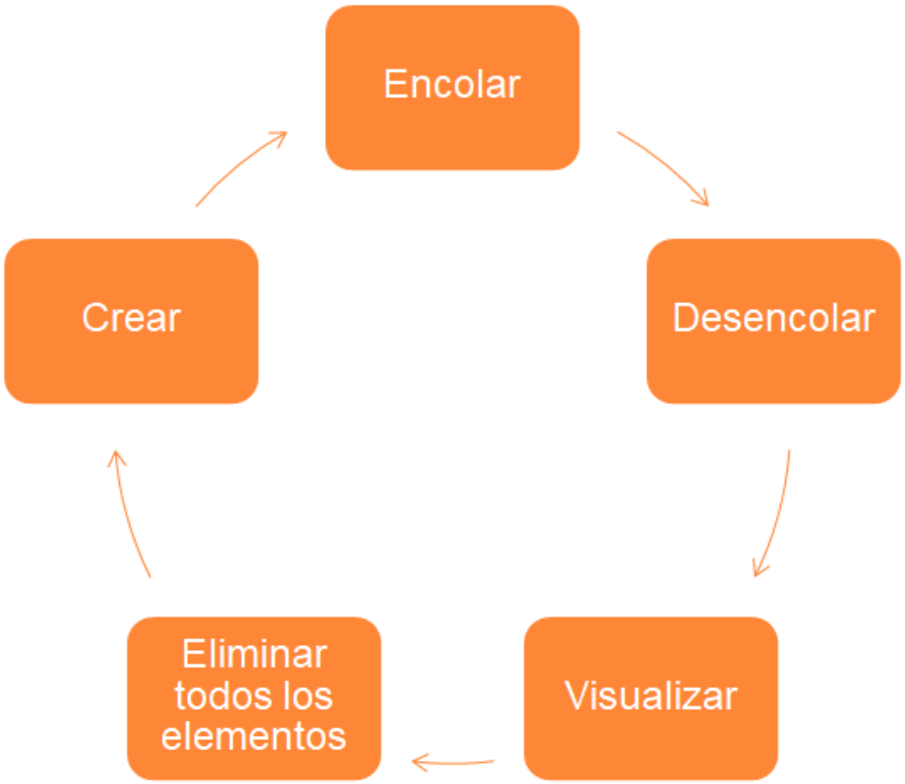
```
struct nodo {
   int dato;
   struct nodo *siguiente;
};
```

Los tipos que definiremos normalmente para manejar colas serán casi los mismos que para manejar listas y pilas, tan sólo cambiaremos algunos nombres:

```
typedef struct _nodo {
   int dato;
   struct _nodo *siguiente;
} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Cola;
```

`tipoNodo` es el tipo para declarar nodos, evidentemente.
`pNodo` es el tipo para declarar punteros a un nodo.
`Cola` es el tipo para declarar colas.

1.8.2.2 Operaciones básicas con colas



a) Crear

```
cola CREAR () {
   cola C;
   C = (tcola *) malloc(sizeof(tcola));
   if (C == NULL)
          error("Memoria insuficiente.");
   C->ant = C->post = (celda *)malloc(sizeof(celda));
   if (C->ant == NULL) error("Memoria insuficiente.");
   C->ant->siguiente = NULL;
return C;
}
```

b) Insertar (encolar)

La inserción en las colas se realiza por la cola de las mismas, es decir, se inserta al final de la estructura.

Para llevar a cabo esta operación únicamente hay que reestructurar un par de punteros, el último nodo debe pasar a apuntar al nuevo nodo (que pasará a ser el último) y el nuevo nodo pasa a ser la nueva cola de la cola.

c) Eliminar (Desencolar)

Se quita de la cola el primer elemento. Para ello, una variable auxiliar apunta al inicio de la cola. Luego el siguiente elemento de la cola se mueve al inicio. Y finalmente se borra (*delete*) la variable auxiliar.

```
/*                      Desencolar elemento

int desencolar( struct cola &q )
{
     int num ;
     struct nodo *aux ;

     aux = q.delante;       // aux apunta al inicio de la cola
     num = aux->nro;
     q.delante = (q.delante)->sgte;
     delete(aux);           // libera memoria a donde apuntaba aux

     return num;
}
```

d) Visualizar

Utilizando un "*while*" se recorre toda la cola, desde la cabeza hasta el último elemento de la misma.

```
/*                          Mostrar Cola
void muestraCola( struct cola q )
{
     struct nodo *aux;
     aux = q.delante;
     while( aux != NULL )
     {
            cout<<"   "<< aux->nro ;
```

```
            aux = aux->sgte;
      }
}
```

e) Eliminar todos los elementos de la cola

Utilizando un "*while*" se recorre toda la cola, desde la cabeza hasta el último elemento de la misma y uno a uno se los borra (*delete*).

```
/*              Eliminar todos los elementos de la Cola
void vaciaCola( struct cola &q)
{
      struct nodo *aux;
      while( q.delante != NULL)
      {
            aux = q.delante;
            q.delante = aux->sgte;
            delete(aux);
      }
      q.delante = NULL;
      q.atras   = NULL;
}
```

1.8.2.3 Ejemplo de cola en C

```
#include <stdlib.h>
#include <stdio.h>

typedef struct _nodo {
   int valor;
   struct _nodo *siguiente;
} tipoNodo;

typedef tipoNodo *pNodo;

/* Funciones con colas: */
void Anadir(pNodo *primero, pNodo *ultimo, int v);
int Leer(pNodo *primero, pNodo *ultimo);

int main()
{
   pNodo primero = NULL, ultimo = NULL;
```

```
   Anadir(&primero, &ultimo, 20);
   printf("Añadir(20)\n");
   Anadir(&primero, &ultimo, 10);
   printf("Añadir(10)\n");
   printf("Leer: %d\n", Leer(&primero, &ultimo));
   Anadir(&primero, &ultimo, 40);
   printf("Añadir(40)\n");
   Anadir(&primero, &ultimo, 30);
   printf("Añadir(30)\n");
   printf("Leer: %d\n", Leer(&primero, &ultimo));
   printf("Leer: %d\n", Leer(&primero, &ultimo));
   Anadir(&primero, &ultimo, 90);
   printf("Añadir(90)\n");
   printf("Leer: %d\n", Leer(&primero, &ultimo));
   printf("Leer: %d\n", Leer(&primero, &ultimo));

   system("PAUSE");
   return 0;
}

void Anadir(pNodo *primero, pNodo *ultimo, int v)
{
   pNodo nuevo;

   /* Crear un nodo nuevo */
   nuevo = (pNodo)malloc(sizeof(tipoNodo));
   nuevo->valor = v;
   /* Este será el último nodo, no debe tener siguiente */
   nuevo->siguiente = NULL;
   /* Si la cola no estaba vacía, añadimos el nuevo a continuación
de ultimo */
   if(*ultimo) (*ultimo)->siguiente = nuevo;
   /* Ahora, el último elemento de la cola es el nuevo nodo */
   *ultimo = nuevo;
   /* Si primero es NULL, la cola estaba vacía, ahora primero
apuntará también al nuevo nodo */
   if(!*primero) *primero = nuevo;
}

int Leer(pNodo *primero, pNodo *ultimo)
{
   pNodo nodo; /* variable auxiliar para manipular nodo */
   int v;      /* variable auxiliar para retorno */

   /* Nodo apunta al primer elemento de la pila */
   nodo = *primero;
   if(!nodo) return 0; /* Si no hay nodos en la pila retornamos 0
*/
   /* Asignamos a primero la dirección del segundo nodo */
   *primero = nodo->siguiente;
   /* Guardamos el valor de retorno */
```

```
    v = nodo->valor;
    /* Borrar el nodo */
    free(nodo);
    /* Si la cola quedó vacía, ultimo debe ser NULL también*/
    if(!*primero) *ultimo = NULL;
    return v;
}
```