5 Tipos Abstractos de Datos

Contenido

1. Introducción. Definición de Tipos Abstractos de Datos.

- ***Abstracción* o *encapsulamiento*:**

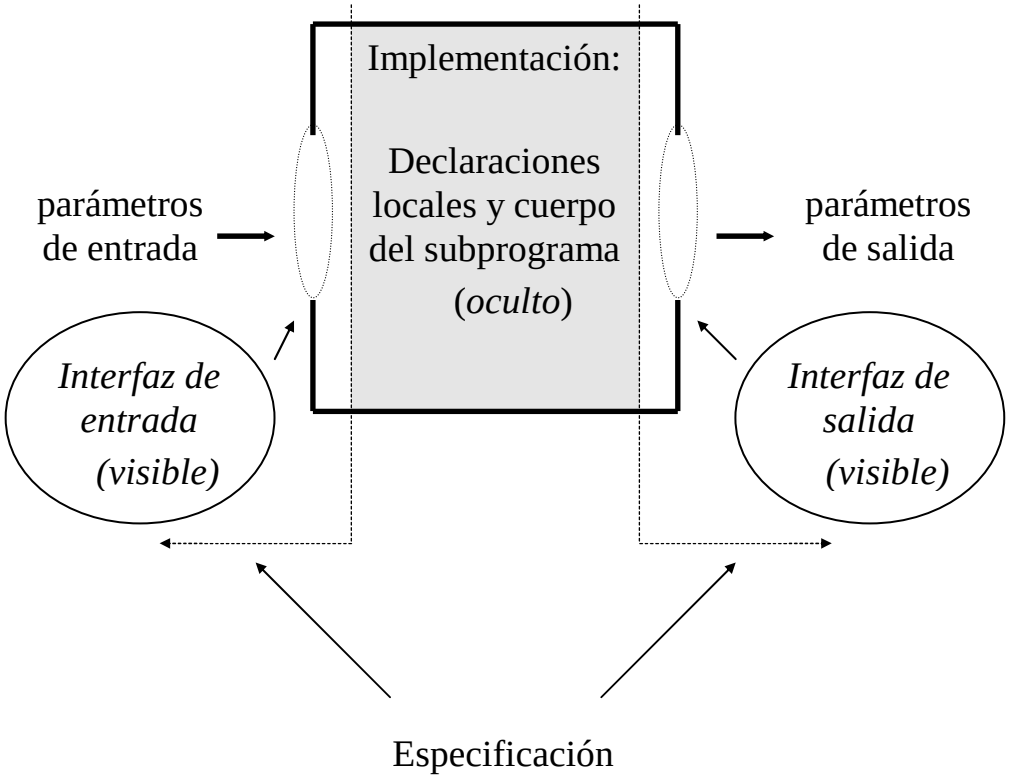
  Separación de la *especificación* de un objeto o algoritmo de su *implementación*, en base a que su *utilización* dentro de un programa sólo debe depender de un interfaz explícitamente definido (la especificación) y no de los detalles de su representación física (la implementación), los cuales están *ocultos*.

- **Ventajas de la abstracción:**

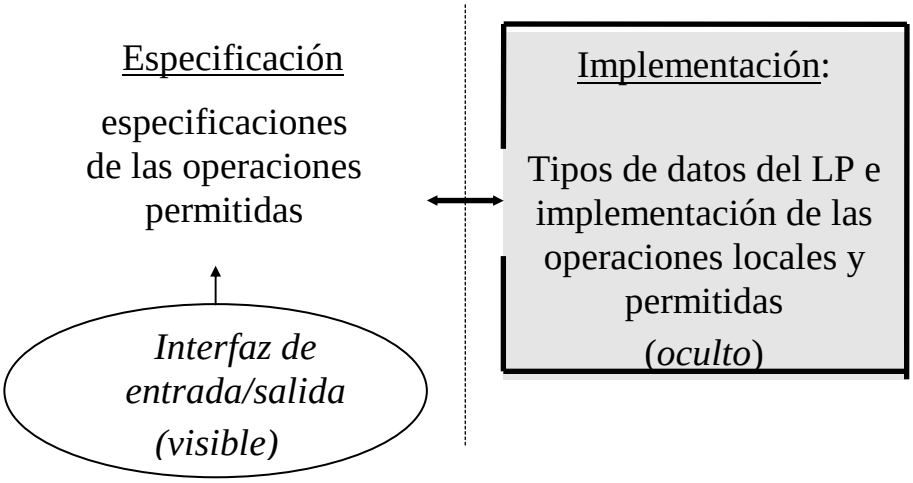
  Establece la *independencia* de QUÉ es el objeto (o QUÉ hace el algoritmo) y de CÓMO está implementado el objeto (o algoritmo), permitiendo la modificación del CÓMO sin afectar al QUÉ, y por lo tanto, sin afectar a los programas que utilizan este objeto o algoritmo.

- **Tipos de abstracciones:**

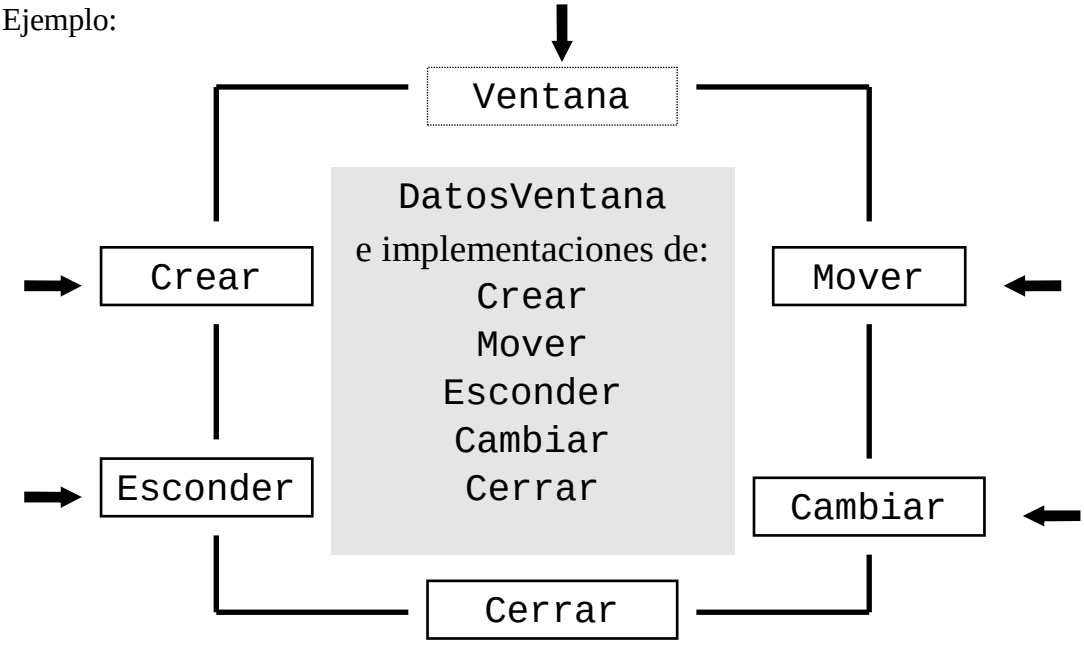
  Abstracción de operaciones. Una serie de operaciones básicas se encapsulan para realizar una operación más compleja. En los lenguajes de programación este tipo de abstracción se logra mediante los *subprogramas*:



  Abstracción de datos. Se encapsula la representación interna de un dato junto con las implementaciones de todas las operaciones que se pueden realizar con ese dato. Esta encapsulación implica que estos datos ocultos sólo pueden modificarse a través de la especificación (o interfaz de e/s). En los lenguajes de programación, la abstracción de datos se logra mediante los tipos de datos que suministra el lenguaje (enteros, reales, arrays, registros, …) y los subprogramas que van a implementar las operaciones permitidas, algunas de cuyas cabeceras formarán su especificación.



- Ejemplo:



Un Tipo Abstracto de Datos (TAD) consta de su **ESPECIFICACIÓN o DEFINICIÓN** (en la que se define la estructura de datos y las operaciones que se realizan sobre ella) que será **independiente** de su **IMPLEMENTACIÓN** (donde se implementan las operaciones descritas en la definición). Puede implementarse de diversas formas siempre que cumplan su definición. Además, la definición nos proporcionará las *operaciones*, las cuales, a su vez, nos determinarán la *utilización* del TAD.

Un programador puede crear un TAD creando una estructura de datos y definiendo e implementando unas operaciones sobre ella. Por ejemplo, para utilizar números complejos podemos definir que un número complejo es una estructura con dos valores y posteriormente definir operaciones sobre ese tipo de dato: creación de un número complejo, suma de complejos, multiplicación, división,...

Una característica importante de un programa que utilice un TAD creado por el programador es que si se cambia cualquier aspecto de la estructura de datos o de la implementación de sus operaciones, el programa debería seguir funcionando correctamente. Es decir, debe haber una independencia entre el programa que utiliza el TAD y la implementación del mismo.

En general, es recomendable que cuando se crea un TAD se implemente en un módulo separado, aprovechando así las ventajas de la compilación separada (reducir tiempo de ejecución, poder reutilizar fácilmente ese TAD en otros programas...). O sea, es conveniente que el TAD esté definido en una biblioteca independiente. En el fichero de cabecera .h de la biblioteca se definen las estructuras de datos del TAD y se declaran los prototipos de las funciones que implementan las operaciones. El fichero .cpp de la biblioteca contendrá el código de las funciones que implementan las operaciones.

2. Algunos TAD típicos.

Hay muchos TAD posibles, pero hay algunos muy típicos y que se suelen utilizar en programación para solucionar multitud de problemas. Algunos de ellos son:

- **Lista:** Es una sucesión de elementos en cierto orden. El número de elementos puede variar y puede haber elementos duplicados. Algunas operaciones que pueden incluirse son: crear y destruir una lista, insertar elementos en ella, recuperar cierto elemento, borrar un elemento, etc.
- **Pila:** Es una estructura LIFO (Last In First Out). Consiste en un conjunto de elementos del que sólo podemos leer el último e insertar en la última posición. Algunas operaciones que pueden incluirse son: crear y destruir una pila, insertar un elemento en la última posición, leer el último elemento o borrar el último elemento. Una pila puede construirse como una lista con las operaciones restringidas.
- **Cola:** Es una estructura FIFO (First In First Out). Consiste en un conjunto de elementos del que sólo podemos leer el primero e insertar en la última posición. Algunas operaciones que pueden definirse son: crear y destruir una cola, insertar un elemento en la última posición, leer el primer elemento o borrar el primer elemento. También una cola puede construirse como una lista con las operaciones restringidas.
- **Árbol:** Estructura con distintos nodos o elemento, tal que cada elemento puede tener a otros elementos como "hijos". Nodos "hoja" son los que no tiene hijos y el nodo "raíz" es el nodo principal o aquel que no tiene "padre". Se llaman árboles binarios si se admite un máximo de dos hijos por nodo. Como operaciones básicas sobre los árboles pueden considerarse las siguientes: crear y destruir un nodo, asignar un nodo como hijo de otro, obtener el padre de un nodo, calcular el número de hijos de un nodo, obtener cierto hijo particular de un nodo, obtener el nodo raíz de un árbol, insertar un hijo a un nodo, podar una rama concreta del árbol, recorrer todos los descendientes de un nodo (en preorden, postorden o inorden)...
- **Grafo:** Es un conjunto de nodos (elementos o vértices) que pueden conectarse entre sí de cualquier manera. A cada conexión entre dos nodos se le llama "arista". Se llaman grafos dirigidos si a las aristas se les supone algún sentido o dirección, de forma que el sentido opuesto no es permitido. Algunas operaciones son: crear y destruir un grafo, crear y destruir un vértice, crear y destruir una arista, recorrer un grafo de un vértice a otro, calcular el camino mínimo entre dos vértices, ...

Los tres primeros son TAD lineales mientras que los dos últimos son TAD no lineales.

No se ha dicho nada sobre lo que se entiende por "elemento". Un elemento puede ser un número, una letra, varios números, ... o cualquier otra estructura de datos.

3. Listas.

Especificación del TAD `Lista`

`TipoLista`

Colección de elementos homogéneos (del mismo tipo: `TipoElemento`*) con una relación LINEAL establecida entre ellos. Pueden estar ordenadas o no con respecto a algún valor de los elementos y se puede acceder a cualquier elemento de la lista.*

Operaciones

```
TipoLista Crear()
void Imprimir(TipoLista lista)
int ListaVacía(TipoLista lista)
void Insertar(TipoLista lista, TipoElemento elem)
void Eliminar(TipoLista lista, TipoElemento elem)
```

Opcionalmente, si la implementación lo requiere, pueden definirse otras operaciones:

```
int ListaLlena(TipoLista lista)
int Longitud(TipoLista lista)
...
```

- Hay que tener en cuenta que la posición de la inserción no se especifica, por lo que dependerá de la implementación. No obstante, cuando la posición de inserción es la misma que la de eliminación, tenemos una subclase de lista denominada ***pila***, y cuando insertamos siempre por un extremo de la lista y eliminamos por el otro tenemos otra subclase de lista denominada ***cola***.
- En el procedimiento `Eliminar` el argumento `elem` podría ser una clave, un campo de un registro `TipoElemento`.

Implementación

La *implementación* o *representación física* puede variar:

Representación mediante arrays

La Lista es un array de tamaño máximo `MaxLista`. Debemos conocer, además, el número de elementos que tiene la lista. Para ello se puede utilizar una variable que inicialmente toma el valor 0 para indicar que la lista está vacía y que se debe incrementar o decrementar convenientemente cuando se insertan o eliminan elementos. Estos elementos se pueden insertar y borrar en cualquier posición de la lista. Para insertar o eliminar elementos podría exigirse que se desplazaran los elementos de modo que no existieran huecos. Si se utilizan arrays estáticos se tiene la desventaja de tener que dimensionar la estructura global de antemano. Si se trata de un array dinámico tiene el problema de tener que reservar un bloque contiguo de memoria dinámica. Si este bloque es muy grande puede que no haya suficiente memoria disponible.

- La implementación con arrays estáticos usará los tipos:

```
/* TIPOS */

typedef int TipoElemento; /* U otro cualquiera */
typedef struct{
        TipoElemento elementos[MaxLista];
        unsigned num_elem; /* num_elem: [0..MaxLista] */
}TipoLista;
```

- Mientras que con arrays dinámicos utilizará:

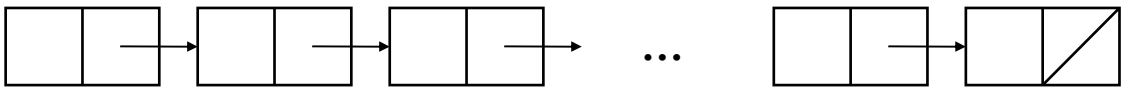
```
/* TIPOS */

typedef int TipoElemento; /* U otro cualquiera */
typedef struct{
        TipoElemento *elementos;
        unsigned num_elem; /* num_elem: [0..] */
}TipoLista;
```

Representación mediante listas enlazadas con punteros

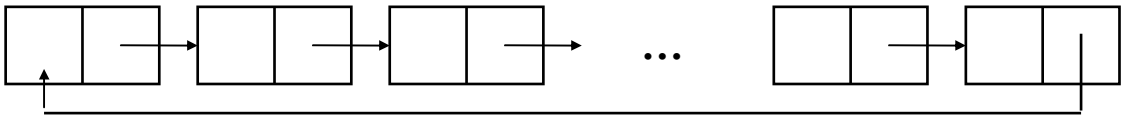
Ventajas:

- se evitan movimientos de datos al insertar y eliminar elementos de la lista.
- Permite dimensionar en tiempo de ejecución (cuando se puede conocer las necesidades de memoria). La implementación de las operaciones son similares a las ya vistas en el tema anterior sobre la lista enlazada de punteros. La implementación completa de una lista sin cabecera y una lista con cabecera se ha incluido en los apéndices A y B respectivamente.



Listas circulares

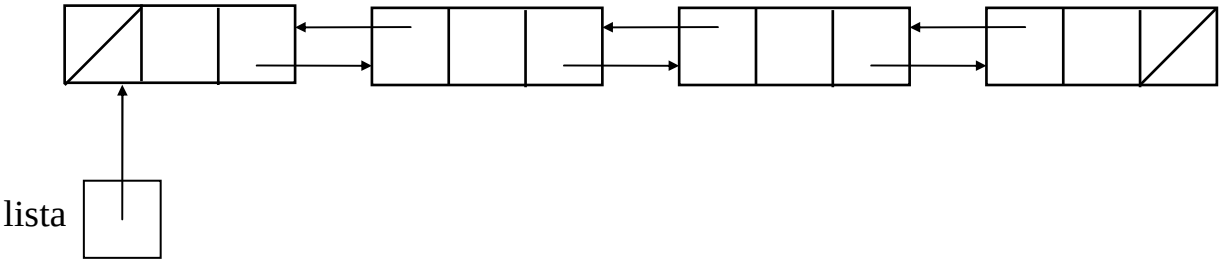
- Son listas en las que el último elemento está enlazado con el primero, en lugar de contener el valor `NULL`. Evidentemente se implementarán con una representación enlazada. Con punteros sería:



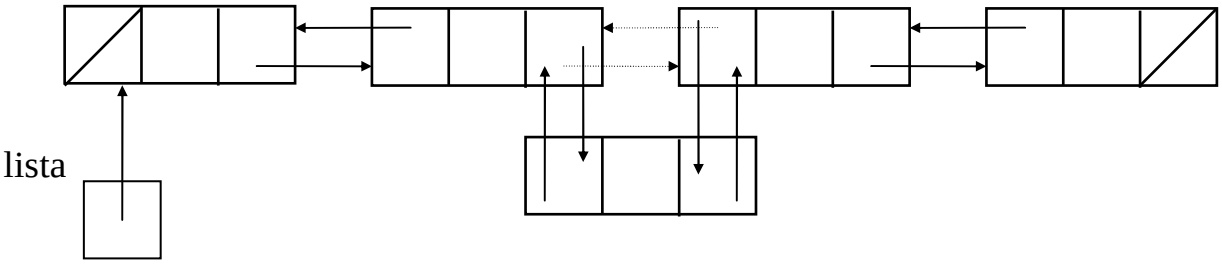
- Una lista vacía la representaremos por el valor `NULL`. Es conveniente guardar en algún sitio el primer o último elemento de la lista circular para detectar el principio o el final de la lista, ya que `NULL` ahora sólo significa que la lista está vacía. Así, las comparaciones para detectar final de lista se harán con el propio puntero de la misma.

Listas doblemente enlazadas

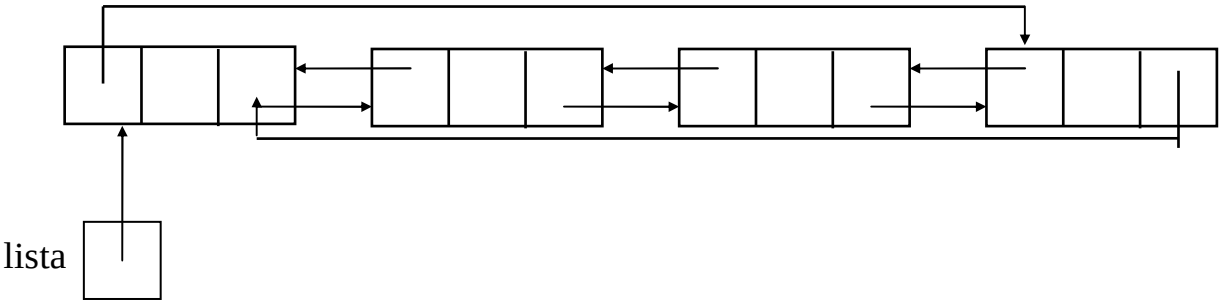
- Son listas en las que cada nodo, además de contener los datos información propios del nodo, contiene un enlace al nodo anterior y otro al nodo siguiente:



- Aparte de que ocupan más memoria (el tamaño de un puntero por cada nodo más), los algoritmos para implementar operaciones para listas dobles son más complicados que para las listas simples porque requieren manejar más punteros. Para insertar un nodo, por ejemplo, habría que cambiar cuatro punteros (o índices si se simulan con un array):



- La ventaja es que pueden ser recorridas en ambos sentidos, con la consiguiente eficiencia para determinados procesamientos.
- Una lista doble también puede ser circular:

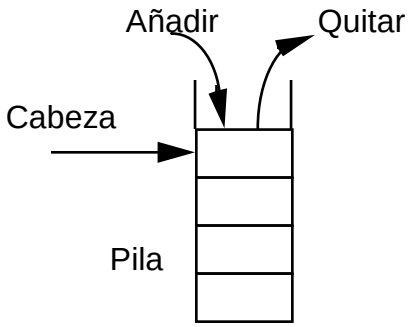


4. Pilas

Especificación del TAD `Pila`

`TipoPila`

Colección de elementos homogéneos (del mismo tipo: `TipoElemento`*) ordenados cronológicamente (por orden de inserción) y en el que sólo se pueden añadir y extraer elementos por el mismo extremo, la cabeza. Es una estructura LIFO (Last In First Out):*



Operaciones

```
TipoPila Crear(void);  /* vacía */
/* Crea una pila vacía */
int PilaVacia(TipoPila pila);
/* Comprueba si la pila está vacía */
int PilaLlena(TipoPila pila);
/* Comprueba si la pila está llena */
void Sacar(TipoPila pila,  TipoElemento& elem);
/* Saca un elemento. No comprueba antes si la pila está vacía */
void Insertar(TipoPila pila, TipoElemento elem);
/* Inserta un elemento en la pila. No comprueba si está llena. */
```

Implementación

Con un array

- Podemos poner los elementos de forma secuencial en el array, colocando el primer elemento en la primera posición, el segundo en la segunda posición, y así sucesivamente. La última posición utilizada será la cabeza, que será la única posición a través de la cual se podrá acceder al array (aunque el array permita el acceso a cualquiera de sus elementos). Para almacenar el valor de la cabeza utilizamos una variable del tipo índice del array (normalmente entera):

```
const int MaxPila = 100; /* Dimension estimada */
/* TIPOS */
   typedef int TipoElemento; /* cualquiera */
   typedef struct{
       int Cabeza; /* Indice */
       TipoElemento elementos[100];
   }TipoPila;
```

- Si el tipo base de los elementos de la pila coincide con el tipo índice o es compatible, podemos utilizar una posición del propio array (`pila[0]` por ejemplo), para guardar el valor de la cabeza:

```
const MaxPila = 100;
const   Cabeza = 0; /* posición con el índice
                       de la cabeza */

/* TIPOS */
    TipoElemento TipoPila[100];
```

- ¡Ojo con las precondiciones de las operaciones `Insertar` y `Sacar`!:

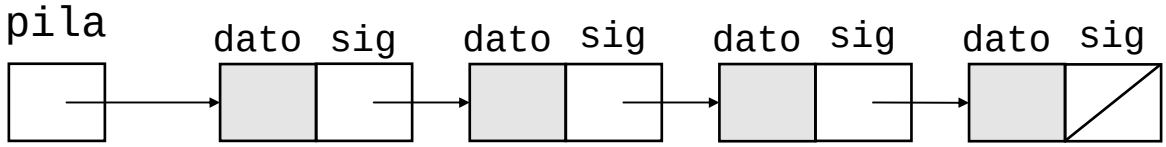
  a) No se puede insertar más elementos en una pila llena.

  b) No se puede sacar elementos de un pila vacía.

  Pueden tenerse en cuenta ANTES de cada llamada a `Insertar` y `Sacar`, o preferiblemente en los propios procedimientos `Insertar` y `Sacar`.

Como una lista enlazada de punteros

- Los nodos de la pila están enlazados con punteros, de forma que se va reservando/liberando memoria según se necesita (por lo que no es necesario implementar la operación `PilaLlena`). El comienzo de la lista enlazada se trata como si fuera la cabeza de la pila: todas las operaciones se realizan al principio de la lista.



- Los tipos necesarios serían:

```
/* TIPOS */
    typedef int TipoElemento; /*o cualquiera */
    typedef struct nodo{
        TipoElemento dato;
        struct nodo *sig;
    }TipoNodo;
    typedef TipoNodo *TipoPuntero;
    typedef TipoPuntero TipoPila;
```

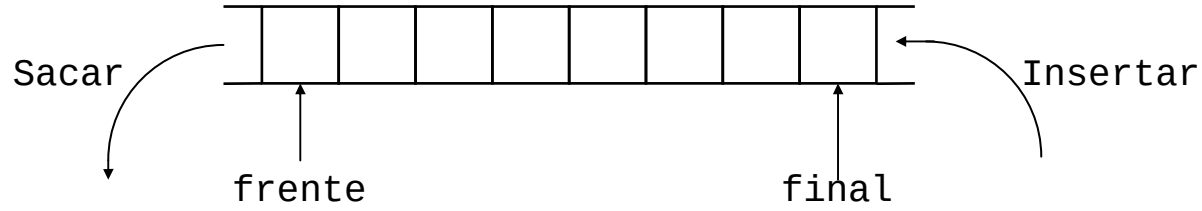
- La operación `Insertar` se corresponderá con la de insertar un nodo al principio de la lista enlazada y la de `Sacar` con la de recuperar el primer nodo de la lista enlazada y eliminarlo después.

5. Colas

Especificación del TAD `Cola`

`TipoCola`

Colección de elementos homogéneos (del mismo tipo: `TipoElemento`*) ordenados cronológicamente (por orden de inserción) y en el que sólo se pueden añadir elementos por un extremo (final) y sacarlos sólo por el otro (frente). Es una estructura FIFO (First In First Out):*



Operaciones

```
TipoCola Crear();
/* Crea y devuelve una cola vacía */
int ColaVacía(Tipocola Cola);
/* Devuelve 1 sólo si "cola" está vacía */
int ColaLlena(TipoCola Cola);
/* Devuelve 1 sólo si "cola" está llena */
int Sacar(Tipocola Cola, TipoElemento& elem);
/* Saca el siguiente elemento del frente y lo pone en "elem" */
int Insertar(TipoCola Cola, TipoElemento elem);
/* Inserta "elem" al final de la cola */
```

- ¡Ojo con las precondiciones de las operaciones `Insertar` y `Sacar`!:

  a) No se puede insertar más elementos en una cola llena.

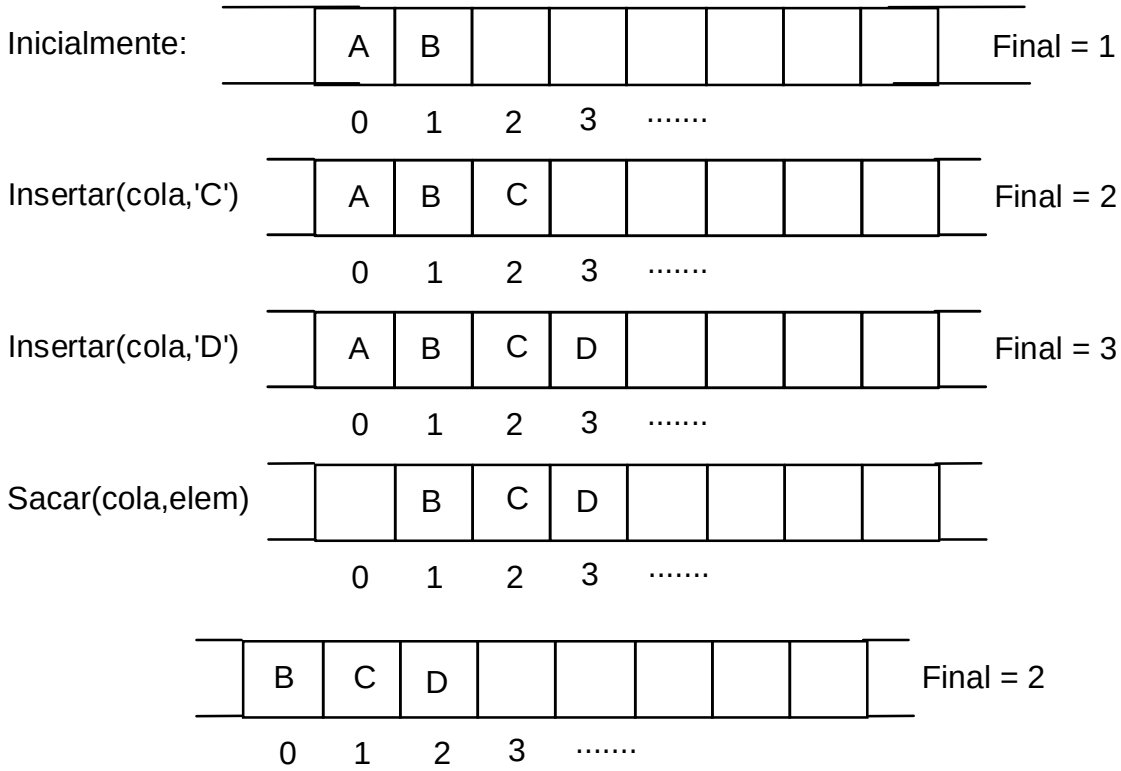
  b) No se puede sacar elementos de un cola vacía.

  Pueden tenerse en cuenta ANTES de cada llamada `Insertar` y `Sacar`, o en los propios procedimientos `Insertar` y `Sacar`.

Implementación

Con un array de frente fijo

- La cola es un array de tamaño `MaxCola` cuyo frente coincide con la primera posición de éste y final inicialmente vale -1. Para añadir elementos simplemente variamos el final. Pero cada vez que sacamos un elemento del frente (de la 1ª posición), tenemos que desplazar el resto una posición en el array para hacer coincidir de nuevo el frente de la cola con la primera posición del array. Así frente siempre vale la primera posición y final indica la última posición ocupada del array. Ejemplo:



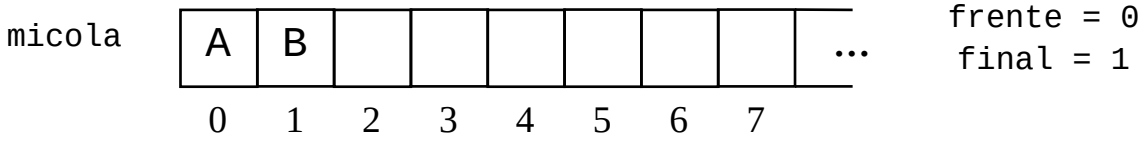
- Ventaja: simplicidad en las operaciones `Crear`, `ColaVacía`, `ColaLlena` y `Insertar`.
- Desventajas:

  a) Las propias de las estructuras estáticas: desaprovechamiento de la memoria.

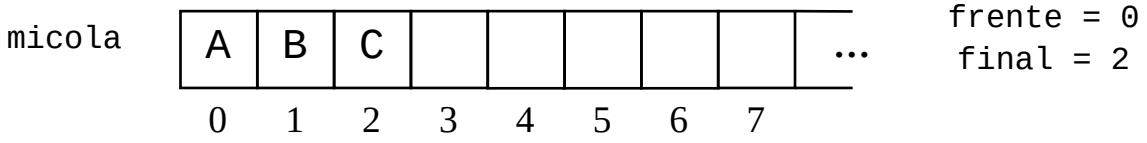
  b) Si la cola va a almacenar muchos elementos o elementos grandes, la operación `Sacar` resulta muy costosa.

Con un array de frente móvil

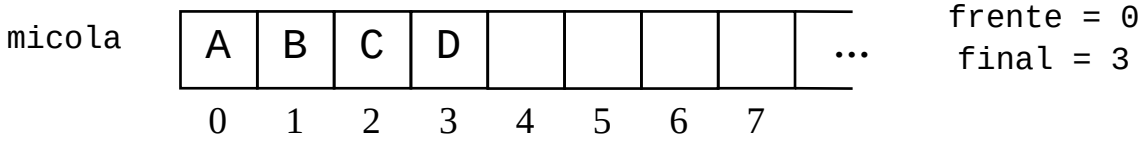
- Llevamos un índice para el frente igual que para el final y permitimos que ambos fluctúen circularmente por un array de rango [`0..MaxCola-1`]:



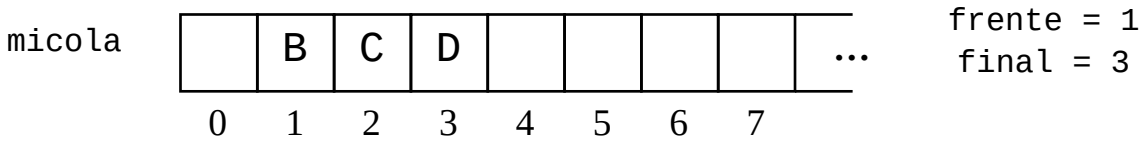
`Insertar(micola, 'C')`



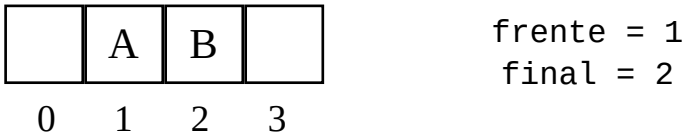
`Insertar(micola, 'D')`



```
Sacar(micola, elem)
```



- Ventaja: la operación `Sacar` es más sencilla y menos costosa.
- Desventaja: hay que tratar el array como una estructura circular, lo que origina el problema de que no podemos diferenciar la situación de cola vacía y cola llena, ya que en ambos casos coincidirían los valores de final y frente. Para comprobar que la lista está vacía se podría utilizar la condición *if (final==frente-1),* pero esta condición se cumpliría igualmente en el caso en que la lista estuviese llena si se ha dado la vuelta por el otro extremo al ser la lista una lista circular.

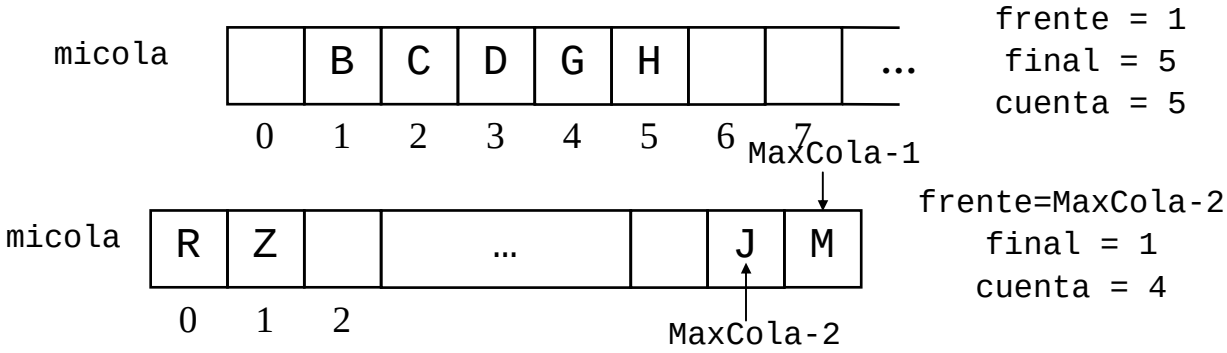


A continuación inserto C y D, con lo que la lista está llena:

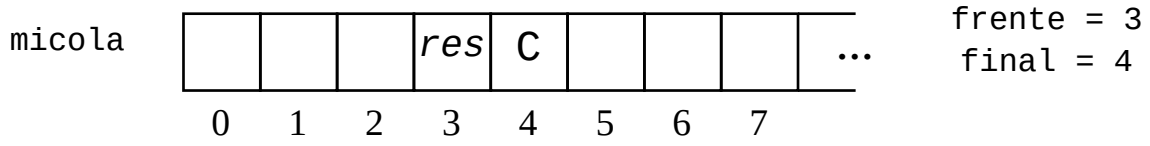
Esto es indistinguible de tener, inicialmente, la lista anterior, y borrar todos los elementos, con lo que estaría la lista vacía y, nuevamente:

Para solucionar este problema tenemos dos posibles implementaciones:

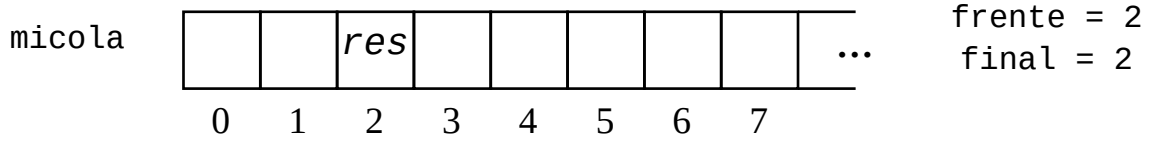
a) Disponer de otra variable, además de `frente` y de `final`, que lleve la cuenta de los elementos de la cola. Si esta variable es cero, la cola está vacía, y si es `Maxcola`, la cola está llena. Ejemplos:



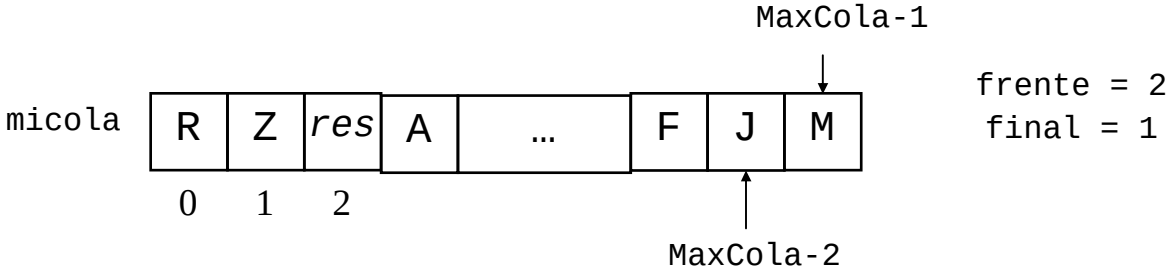
b) Hacer que `frente` apunte a la casilla del array que precede a la del elemento frente de la cola, en vez de a la del propio elemento frente, la cual tendrá un valor *`reservado`*[*] que también fluctúa por el array:



Así, si `frente` es igual a `final`, ello significa que la cola está vacía[†]:



La cola está llena si el siguiente espacio disponible (`final+1`) está ocupado por el valor especial *`reservado`* e indicado por `frente`:



- La implementación con la primera opción usará los tipos:

```
/* TIPOS */

typedef int TipoElemento /* U otro cualquiera */
typedef struct{
        TipoElemento elementos[MaxCola];
        unsigned frente; /* Frente: [0..MaxCola-1] */
        int final; /* Final: [0..MaxCola-1] */
        unsigned cuenta; /* Cuenta: [0..MaxCola] */
}TipoCola;
```

- Mientras que la segunda opción utilizará:

```
/* TIPOS */

typedef int TipoElemento /* U otro cualquiera */
typedef struct{
```

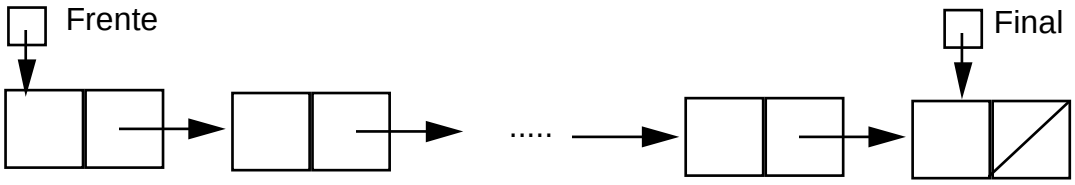
[*] ¿Cuál será su posición inicial? -> en 0. Inicialmente, *frente* y *final* valen 0 para indicar que la lista está vacía

[†] ¿Cuánto valdrá `final` inicialmente? -> 0, al igual que *frente*

```
        TipoElemento elementos[MaxCola];
        unsigned frente; /* Frente: [0..MaxCola-1] */
        int final; /* Final: [0..MaxCola-1] */
}TipoCola;
```

Utilizando listas enlazadas con punteros

- Dos variables de tipo puntero, `frente` y `final`, van a apuntar a los nodos que contienen los elementos frente y final respectivamente.



- Para sacar elementos de la cola podemos utilizar un algoritmo similar al de borrar el primer nodo de una lista, considerando que `frente` apunta al mismo.
- Para añadir un elemento a la cola realizamos tres pasos:

  1) Creamos un nuevo nodo para el elemento a añadir.

  2) Insertamos el nuevo nodo al final de la cola.

  3) Actualizamos `final`.
- Los tipos necesarios serán:

```
/* TIPOS */

typedef int TipoElemento; /* O cualquiera */

typedef struct nodo{
        TipoElemento valor;
        struct nodo *sig;
}TipoNodo;

typedef TipoNodo *TipoPuntero;

typedef struct{
        TipoPuntero frente;
        TipoPuntero final;
}TipoCola;
```

- Ventaja: aprovechamiento de la memoria por el uso de punteros.
- Desventajas: las propias de punteros.

APÉNDICE A: Tipo Lista Sin Cabecera

```
/*********************************************************/
/* Ejemplo de programa para manejar una:                 */
/*   LISTA SIMPLEMENTE ENLAZADA SIN CABECERA,            */
/*   como Estructura de Datos Dinámica (con punteros).   */
/* Incluye: Primitivas de la lista y programa de prueba. */
/*          Todo en el mismo fichero.                    */
/*********************************************************/

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

```
#include<conio.h>

/* Declaración de tipos */
struct element {
  long num;
  struct element *sig;
};

typedef struct element *lista;

/*- PRIMITIVAS de una LISTA SIMPLEMENTE ENLAZADA SIN CABECERA -*/

/*****************************************************************/
/* Inicializa una lista simplemente enlazada y sin cabecera.   */
void inicializar_lista(lista& list){
  list=NULL;
}

/*****************************************************************/
/* Devuelve TRUE si la lista está vacía.                         */
int lista_vacia(lista list){
  if (list) return 0;
  return 1;
}

/*****************************************************************/
/* Devuelve la longitud de la lista list (núm. de elementos).  */
long long_lista(lista list){
  long i=0;

  while (list) {
    list=list->sig;
    i++;
  }
  return i;
}
/*****************************************************************/
/* Inserta el elemento e en la posición pos de la lista list.  */
/* Si pos<=1 inserta en la primera posición.                     */
/* Si pos>longitud_lista, inserta en la última posición.       */
/* Devuelve -1 si no hay memoria suficiente para la inserción. */
int insertar_pos (lista& list, struct element e, long pos){
  lista p, anterior, L=list;
  long i=1;

  if ((p=(struct element *) malloc(sizeof(struct element))) == NULL)
     return -1;
  *p=e;

  if (pos<=1 || lista_vacia(list)){
    /* Hay que insertar en la posición 1 */
    list=p;
    p->sig=L;
    return 0;
  }

  while (L && i<pos){
    anterior=L;
    L=L->sig;
    i++;
  }

  /* Insertar elemento apuntado por p, entre anterior y L */
  anterior->sig=p;
```

```
  p->sig=L;
  return 0;
}

/*************************************************************/
/* Inserta el elemento e en la lista list por orden del campo num */
/* Devuelve -1 si no hay memoria suficiente para la inserción. */
int insertar_orden (lista& list, struct element e){
  lista p, anterior, L=list;
  unsigned i=1;

  if ((p=(struct element *) malloc(sizeof(struct element))) == NULL)
     return -1;
  *p=e;

  if (lista_vacia(list) ||e.num<=list->num){
    /* Si hay que insertar antes del primero o la lista está vacía */
    list=p;
    p->sig=L;
    return 0;
  }

  while (L && e.num>L->num){
    anterior=L;
    L=L->sig;
    i++;
  }

  anterior->sig=p;
  p->sig=L;
  return 0;
}

/*************************************************************/
/* Borra el elemento en la posición pos de la lista list.      */
/* Si la pos=1 borra el primer elemento.                       */
/* Devuelve -1 si no existe la posición: pila vacía o pos>long */
int borrar_elemento(lista& list,long pos){
  lista anterior, L=list;
  long i=1;

  if (lista_vacia(list) || pos<1)
     return -1;

  if (pos==1) { /* Tratamiento particular para borrar elemento 1 */
     (list) = (list)->sig;
     free(L);
     return 0;
  }

  while (L && i<pos){
    anterior=L;
    L=L->sig;
    i++;
  }
  if (L){ /* OJO: Aqui no vale decir: if (i==pos)... */
     /*  Borrar  elemento  apuntado  por  L,  teniendo  un  puntero  al
anterior */
     anterior->sig=L->sig;
     free(L);
     return 0;
  }
  return -1; /* La lista tiene menos de pos elementos */
}
```

```
/*************************************************************/
/* Libera la memoria ocupada por toda la lista.              */
void liberar_lista(lista& list){
  while (list) {
    borrar_elemento (list,1);
  }
}

/*************************************************************/
/* Devuelve en e, el elemento que está en la posición pos    */
/* de la lista list.                                         */
/* Si no existe esa posición, devuelve -1. En otro caso 0.   */
int leer_element (lista list, long pos, struct element& e){
  long i=1;

  if (lista_vacia(list) || pos<1)
     return -1;

  while (list && i<pos) {
    list=list->sig;
    i++;
  }
  if (list){
     e=*list;
     return 0;
  }
  return -1;
}
/*************************************************************/
/* Devuelve la posición de la primera ocurrencia del elemento  */
/* e en la lista list, a partir de la posición pos (inclusive).*/
/* Esta ocurrencia será considerada por el campo num.        */
/* Devuelve 0 si no ha sido encontrada.                      */
/* Con esta función, cambiando pos, podremos encontrar TODAS   */
/* las ocurrencias de un elemento en la lista.               */
long posic_lista(lista list, struct element e, long pos){
  long i=1;

  while (list && i<pos) { /* Ir a la posición pos */
    list=list->sig;
    i++;
  }
  if (!list) return 0;

  while (list && e.num!=list->num) { /* Intentar encontrar el elemento
*/
    list=list->sig;
    i++;
  }
  if (!list) return 0;
  return i;
}

/*************************************************************/
/* Actualiza la posición pos de la lista list con el elemento e*/
/* Devuelve -1 si no existe esa posición.                    */
int actualiza_lista(lista list, struct element e, long pos){
  long i=1;

  if (pos<1) return -1;   /* Posición no válida   */
  while (list && i<pos) { /* Ir a la posición pos */
    list=list->sig;
    i++;
```

```
  }
  if (!list) return -1; /* Posición no existe */

  list->num=e.num; /* Actualización */
  return 0;
}
/* FIN de las PRIMITIVAS de la LISTA SIMPLEMENTE ENLAZADA SIN CABECERA
*/


/*************************************************************/
/* Imprime un elemento e por la Salida estándard.            */
void Imprimir_elemento (struct element e){
  printf("%li",e.num);
}

/*************************************************************/
/* Muestra todos los elementos de la lista list por su orden.  */
void mostrar_lista(lista list){
  struct element e;
  unsigned i=1,tama=long_lista(list);

  while (i<=tama) {
    printf("\nElemento %u: ",i);
    leer_element(list,i++,e);
    Imprimir_elemento(e);
  }
}
/*************************************************************/
/* Muestra un menu de opciones.                              */
void menu(void){
  puts("\n\t\t***** MENU *****\n");
  puts("\tS. Destruir lista y SALIR.");
  puts("\t1. Inicializar lista.");
  puts("\t2. Insertar por orden.");
  puts("\t3. Insertar por posición.");
  puts("\t4. Mostrar lista.");
  puts("\t5. Borrar elemento.");
  puts("\t6. Longitud lista");
  puts("\t7. Ver si está vacía.");
  puts("\t8. Ver elemento n-ésimo.");
  puts("\t9. Posición de un elemento a partir de una dada.");
  puts("\t0. Modificar un elemento en una posición.");
  printf("\n\tOpcion: ");
}


/*************************************************************/
void main(){
  lista list;
  struct element e;
  char opcion='x';
  long pos;

  inicializar_lista(list);

  while (opcion!='S' && opcion!='s'){
     menu();
     opcion=getche();

     switch (opcion) {
       case 's':
       case 'S':liberar_lista(list);
                break;
```

```
       case '1':liberar_lista(list);
                inicializar_lista(list);
                break;

       case '2':printf("\nDame número: ");
                scanf("%li",&(e.num));
                if (insertar_orden(list,e))
                   puts("\n\aERROR: No existe memoria suficiente.");
                break;

       case '3':printf("\nDame número: ");
                scanf("%li",&(e.num));
                printf("\nDame la posición: ");
                scanf("%li",&pos);
                if (insertar_pos(list,e,pos))
                   puts("\n\aERROR: No existe memoria suficiente.");
                break;

       case '4':printf("\n----------------");
                mostrar_lista(list);
                puts("\n----------------");
                getch();
                break;

       case '5':printf("\nDame posición a borrar: ");
                scanf("%li",&pos);
                if (borrar_elemento(list,pos))
                   puts("\n\aERROR: No existe esa posición.");
                break;

       case '6':printf("\n- Longitud: %u.",long_lista(list));
                break;


       case '7':if (lista_vacia(list))
                     puts("\n- Lista VACIA.");
                else puts("\n- Lista NO VACIA.");
                break;

       case '8':printf("\nDame posición a leer: ");
                scanf("%li",&pos);
                if (leer_element(list,pos,e))
                   puts("\n\aERROR: No existe esa posición.");
                else
                   printf("\n- Elemento %li: %li.\n",pos,e.num);
                break;

       case '9':printf("\nDame número para buscar su posición: ");
                scanf("%li",&(e.num));
                printf("\nDame la posición a partir de la que buscar:
");
                scanf("%li",&pos);
                if (!(pos=posic_lista(list,e,pos)))
                   puts("\n\aERROR: No existe ese elemento a partir de
esa posición.");
                else
                   printf(
               "\n-La primera posición encontrada de ese elemento es:
%i",
                   pos);
              break;

       case '0':printf("\nDame número para actualizar: ");
```

```
                scanf("%li",&(e.num));
                printf("\nDame la posición a modificar: ");
                scanf("%li",&pos);
                if (actualiza_lista(list,e,pos))
                   puts("\n\aERROR: No existe esa posición. No se ha
modificado.");
                break;
     }
  }

  puts("\n\t***** FIN *****");
}
```

APÉNDICE B: Tipo Lista Con Cabecera

Ahora, implementamos una lista CON cabecera y además utilizando las ventajas de la compilación separada, es decir, separamos la implementación de las primitivas de la lista de la implementación del programa que utiliza la lista. Así, tenemos 3 ficheros:

`LSTCCLIB.H` Contiene los tipos de datos y las cabeceras de las primitivas del tipo lista con cabecera. Es utilizado para incluirlo con #include en los siguientes dos ficheros.

`LSTCCLIB.C` Contiene la implementación de las primitivas de la lista.

`PRUCC.C` Programa de prueba de la lista con cabecera. Contiene la función `main()`.

```
/*********************************************************/
/* Fichero: LSTCCLIB.H                                   */
/* Tipos de datos y cabeceras de las PRIMITIVAS para una   */
/*   LISTA SIMPLEMENTE ENLAZADA CON CABECERA.            */
/*********************************************************/
/* Declaración del tipo base de la lista */
struct element {
  long num;
};

/*- PRIMITIVAS de una LISTA SIMPLEMENTE ENLAZADA CON CABECERA -*/

/* Tipos de la lista: */
struct celda{ /* Tipo de cada celda de la lista */
  struct element elem;
  struct celda   *sig;
};

typedef struct celda *lista; /* Tipo lista: Puntero a celta */

int  Inicializar_lista(lista& list);
int  Lista_vacia(lista list);
int  Insertar_pos (lista ant, struct element e, long pos);
int  Insertar_orden (lista ant, struct element e);
int  Borrar_elemento(lista ant, long pos);
long Long_lista(lista list);
void Vaciar_lista(lista list);
void Destruir_lista(lista& list);
int  Leer_element (lista list, long pos, struct element& e);
long Posic_lista(lista list, struct element e, long pos);
int  Actualiza_lista(lista list, struct element e, long pos);
```

```
/*********************************************************/
/* Fichero: LSTCCLIB.C                                   */
/* PRIMITIVAS para el manejo y control de una:           */
/*   LISTA SIMPLEMENTE ENLAZADA CON CABECERA.            */
/* La declaración del tipo elemento (struct element) y   */
/* otros tipos están en el fichero lstcclib.h            */

/*********************************************************/
#include<stdlib.h>
#include"lstcclib.h"

/*********************************************************/
/* Inicializa una lista simplemente enlazada y CON cabecera.   */
/* Devuelve -1 en caso de ERROR.                         */
int Inicializar_lista(lista& list){
  if (((list)=(struct celda *) malloc(sizeof(struct celda))) == NULL)
     return -1;
  (list)->sig=NULL;
  return 0;
}

/*********************************************************/
/* Devuelve TRUE si la lista está vacía.                 */
int Lista_vacia(lista list){
  if (list->sig) return 0;
  return 1;
}

/*********************************************************/
/* Inserta el elemento e en la posición pos de la lista ant.  */
/* Si pos<=1 inserta en la primera posición.             */
/* Si pos>longitud_lista, inserta en la última posición.       */
/* Devuelve -1 si no hay memoria suficiente para la inserción. */
int Insertar_pos (lista ant, struct element e, long pos){
  lista p, L=ant->sig;
  long i=1;

  if ((p=(struct celda *) malloc(sizeof(struct celda))) == NULL)
     return -1;
  p->elem=e;

  while (L && i<pos){ /* Hallar posición en la que insertar */
    ant=L;
    L=L->sig;
    i++;
  }

  ant->sig=p; /* Insertar elemento apuntado por p, entre anterior y L
*/
  p->sig=L;
  return 0;
}

/*********************************************************/
/* Inserta el elemento e en la lista ant ordenadamente por     */
/* el campo elem.num                                     */
/* Devuelve -1 si no hay memoria suficiente para la inserción. */
int Insertar_orden (lista ant, struct element e){
  lista p, L=ant->sig;
  unsigned i=1;

  if ((p=(struct celda *) malloc(sizeof(struct celda))) == NULL)
     return -1;
```

```
  p->elem=e;

  while (L && e.num>L->elem.num){ /* Hallar posición en la que
insertar */
    ant=L;
    L=L->sig;
    i++;
  }


  ant->sig=p; /* Insertar elemento apuntado por p, entre anterior y L
*/
  p->sig=L;
  return 0;
}

/*************************************************************/
/* Borra el elemento en la posición pos de la lista ant.     */
/* Si la pos=1 borra el primer elemento.                     */
/* Devuelve -1 si no existe la posición:                     */
/*      pila vacía, pos<1 o pos>long                         */
int Borrar_elemento(lista ant, long pos){
  lista L=ant->sig;
  long i=1;

  if (Lista_vacia(ant) || pos<1)
     return -1; /* Posición NO válida */

  while (L && i<pos){ /* Situarse en la posición a borrar */
    ant=L;
    L=L->sig;
    i++;
  }

  if (L){
     /* Borrar elemento apuntado por L, teniendo un puntero al
anterior */
     ant->sig=L->sig;
     free(L);
     return 0;
  }
  return -1; /* La lista tiene menos de pos elementos */
}

/*************************************************************/
/* Devuelve la longitud de la lista list (núm. de elementos). */
long Long_lista(lista list){
  long i=0;

  list=list->sig;
  while (list) {
    list=list->sig;
    i++;
  }
  return i;
}

/*************************************************************/
/* Vacía la lista totalmente, dejándola inicializada.        */
/* Tras esta operación NO es necesario inicializarla.        */
void Vaciar_lista(lista list){
  while (Borrar_elemento (list,1) != -1);
}
```

```
/*************************************************************/
/* Destruye la lista totalmente, liberando toda la memoria.  */
/* Tras esto ES necesario Inicializar la lista para reusarla.*/
void Destruir_lista(lista& list){
  Vaciar_lista(list);
  free(list); /* Liberar la cabecera */
  list=NULL;
}

/*************************************************************/
/* Devuelve en e, el elemento que está en la posición pos    */
/* de la lista list.                                         */
/* Si no existe esa posición, devuelve -1. En otro caso 0.   */
int Leer_element (lista list, long pos, struct element& e){
  long i=1;

  if (Lista_vacia(list) || pos<1)
     return -1;

  list=list->sig; /* Nos saltamos la cabecera */
  while (list && i<pos) { /* Localizar la posición pos */
    list=list->sig;
    i++;
  }
  if (list){
     e=list->elem;
     return 0;
  }
  return -1;
}

/*************************************************************/
/* Devuelve la posición de la primera ocurrencia del elemento*/
/* e en la lista list, a partir de la posición pos (inclusive).*/
/* Esta ocurrencia será considerada por el campo elem.num    */
/* Devuelve 0 si no ha sido encontrada.                      */
/* Con esta función, cambiando pos, podremos encontrar TODAS */
/* las ocurrencias de un elemento en la lista.               */
long Posic_lista(lista list, struct element e, long pos){
  long i=1;

  list=list->sig; /* Saltar cabecera */
  while (list && i<pos) { /* Ir a la posición pos */
    list=list->sig;
    i++;
  }
  if (!list) return 0; /* No existe posición pos, luego... */

  while (list && e.num!=list->elem.num) { /* Intentar encontrar el
elemento */
    list=list->sig;
    i++;
  }
  if (!list) return 0; /* No encontrado */
  return i;            /* Encontrado en la posición i */
}

/*************************************************************/
/* Actualiza la posición pos de la lista list con el elemento e*/
/* Devuelve -1 si no existe esa posición.                    */
int Actualiza_lista(lista list, struct element e, long pos){
  long i=1;
```

```
  if (Lista_vacia(list) || pos<1) return -1;    /* Posición no válida
*/

  list=list->sig; /* Saltar cabecera */
  while (list && i<pos) { /* Ir a la posición pos */
    list=list->sig;
    i++;
  }
  if (!list) return -1; /* Posición no existe */

  list->elem=e; /* Actualización */
  return 0;
}

/* FIN de las PRIMITIVAS de la LISTA SIMPLEMENTE ENLAZADA CON CABECERA
*/


/*************************************************************/
/* Fichero: PRUCC.C                                          */
/* Ejemplo de programa para manejar una:                     */
/*   LISTA SIMPLEMENTE ENLAZADA CON CABECERA.                */
/* Utiliza la librería lstcclib.h                            */
/*************************************************************/
#include<stdio.h>
#include<conio.h>
#include"lstcclib.h"

/*************************************************************/
/* Imprime un elemento e por la Salida estándar.             */
void Imprimir_elemento (struct element e){
  printf("%li",e.num);
}

/*************************************************************/
/* Muestra todos los elementos de la lista list por su orden.  */
void mostrar_lista(lista list){
  struct element e;
  unsigned i=1,tama=Long_lista(list);

  while (i<=tama) {
    printf("\nElemento %u: ",i);
    Leer_element(list,i++,e);
    Imprimir_elemento(e);
  }
}

/*************************************************************/
/* Muestra un menu de opciones.                              */
void menu(void){
  puts("\n\t\t***** MENU *****\n");
  puts("\tS. Destruir lista y SALIR.");
  puts("\t1. Inicializar lista.");
  puts("\t2. Insertar por orden.");
  puts("\t3. Insertar por posición.");
  puts("\t4. Mostrar lista.");
  puts("\t5. Borrar elemento.");
  puts("\t6. Longitud lista");
  puts("\t7. Ver si está vacía.");
  puts("\t8. Ver elemento n-ésimo.");
  puts("\t9. Posición de un elemento a partir de una dada.");
  puts("\t0. Modificar un elemento en una posición.");
  printf("\n\tOpcion: ");
}
```

```
/************************************************************/
void main(){
  lista list;
  struct element e;
  char opcion='x';
  long pos;

  Inicializar_lista(list);

  while (opcion!='S' && opcion!='s'){
     menu();
     opcion=getche();

     switch (opcion) {
       case 's':
       case 'S':Destruir_lista(list);
      break;

       case '1':Vaciar_lista(list);
      break;

       case '2':printf("\nDame número: ");
      scanf("%li",&(e.num));
      if (Insertar_orden(list,e))
         puts("\n\aERROR: No existe memoria suficiente.");
      break;


       case '3':printf("\nDame número: ");
      scanf("%li",&(e.num));
      printf("\nDame la posición: ");
      scanf("%li",&pos);
      if (Insertar_pos(list,e,pos))
         puts("\n\aERROR: No existe memoria suficiente.");
      break;

       case '4':printf("\n----------------");
      mostrar_lista(list);
      puts("\n----------------");
      getch();
      break;

       case '5':printf("\nDame posición a borrar: ");
      scanf("%li",&pos);
      if (Borrar_elemento(list,pos))
         puts("\n\aERROR: No existe esa posición.");
      break;

       case '6':printf("\n- Longitud: %u.",Long_lista(list));
      break;

       case '7':if (Lista_vacia(list))
           puts("\n- Lista VACIA.");
      else puts("\n- Lista NO VACIA.");
      break;

       case '8':printf("\nDame posición a leer: ");
      scanf("%li",&pos);
      if (Leer_element(list,pos,e))
         puts("\n\aERROR: No existe esa posición.");
      else
         printf("\n- Elemento %li: %li.\n",pos,e.num);
```

```
      break;

      case '9':printf("\nDame número para buscar su posición: ");
       scanf("%li",&(e.num));
       printf("\nDame la posición a partir de la que buscar: ");
       scanf("%li",&pos);
       if (!(pos=Posic_lista(list,e,pos)))
          puts(
            "\n\aERROR:  No  existe  ese  elemento  a  partir  de  esa
posición.");
       else
          printf(
       "\n-La primera posición encontrada de ese elemento es: %i",
                pos);
       break;

       case '0':printf("\nDame número para actualizar: ");
       scanf("%li",&(e.num));
       printf("\nDame la posición a modificar: ");
       scanf("%li",&pos);
       if (Actualiza_lista(list,e,pos))
          puts("\n\aERROR:  No  existe  esa  posición.  No  se  ha
modificado.");
       break;
     }
  }

  puts("\n\t***** FIN *****");
}
```

APÉNDICE C

TAD **Cola** (* representación con array circular sin cuenta *)

```
#include <stdio.h>
const int MaxCola = 100; /* tamanyo maximo de la cola */
/* TIPOS */
typedef int TipoElemento; /* U otro cualquiera */
typedef struct{
      TipoElemento elementos[100];
      unsigned int frente; /* Frente: [0..MaxCola-1] */
      unsigned int final; /*  Final:  [0..MaxCola-1] */
}TipoCola;

TipoCola Crear(void);
int ColaVacia(TipoCola cola);
int ColaLlena(TipoCola cola);
int Sacar(TipoCola& cola, TipoElemento& elem);
int Meter(TipoCola& cola, TipoElemento elem);

TipoCola Crear(){
  /* Crea una cola vacía */
  TipoCola cola;

  cola.frente = MaxCola-1;  /* precede al primero*/
```

```
  cola.final  = MaxCola-1;  /* vacia */
  return(cola);
}

int ColaVacia(TipoCola cola){
  return(cola.final == cola.frente);
}

int ColaLlena(TipoCola cola){
  if (cola.final==MaxCola-1)
      return(!cola.frente);
  else
      return(cola.final+1 == cola.frente);
}

int Sacar(TipoCola& cola,TipoElemento& elem){
/* Saca un elemento de la cola si no está vacía. Sólo si está
   vacía ANTES de la operación, se devuelve 1, si no, 0" */

  int esta_vacia;

  esta_vacia = ColaVacia(cola);
  if (!esta_vacia){
      if (cola.frente==MaxCola-1) /* Al final */
          cola.frente=0;
      else
          cola.frente++;
      elem = cola.elementos[cola.frente];
  }
  return(esta_vacia);
}

int Meter (TipoCola& cola,TipoElemento elem){
/* Mete un elemento en la cola si no está llena. Sólo si está
   llena ANTES de la operación, se devuelve 1, si no, 0*/

  int esta_llena;
  esta_llena=ColaLlena(cola);

  if (!esta_llena){
      if (cola.final==MaxCola-1) /* Al final */
          cola.final=0;
      else
          cola.final++;
      cola.elementos[cola.final] = elem;
  }
  return(esta_llena);
}
```

```
void main(){
  TipoCola micola;
  int estado=0;
  int i;
  TipoElemento mielem;

  micola=Crear();

  for (i=1; i<=10 && !estado; i++)
      estado=Meter(micola,i);

  for (i=1; i<=5 && !estado; i++){
      estado=Sacar(micola,mielem);
      printf("%d ", mielem);
  }
}
```