- **ABSTRACCION**
- **Tipo de Dato Abstracto - TDA**
- **TDA Matriz**
- **TDA Pila**
- **TDA COLA**

ABSTRACCIÓN

Desde el nacimiento de la informática, los programadores han tenido que enfrentarse al problema de la COMPLEJIDAD.
Una técnica para tratar la complejidad es abstraernos de ella.
Somos incapaces de dominar la totalidad de objetos complejos, para comprenderlos ignoramos los detalles no esenciales, tratando en su lugar el modelo ideal del objeto, centrándonos en sus aspectos esenciales.
La abstracción es la capacidad de encapsular y aislar la información del diseño y la ejecución.
Se construye un modelo que es más sencillo que la realidad que se imita, dando importancia a los aspectos con los que se quiere interactuar.
Por ejemplo, un mapa imita un territorio real, pero no es lo mismo un mapa de carreteras, que uno topográfico o con división política, por lo tanto la elección dependerá de la interacción que se necesite.
Un programa es la descripción abstracta de un procedimiento o fenómeno que existe. Muchas veces imita una acción humana, otras reproduce o simula un hecho físico.
Un lenguaje de programación se utiliza para representar una abstracción del mundo real en sentido doble:

- Para describir en forma abstracta una acción o hecho físico.
- Para describir el comportamiento de la computadora, simplificando con operadores y variables circuitos de memoria, con números decimales los valores binarios representados en dichos circuitos.

Historia de la abstracción

Subprogramas: procedimientos y funciones
Proporcionaron la primera posibilidad de ocultamiento de la información. Un programador desarrolla un procedimiento, y otros, conociendo para qué sirve, lo pueden invocar sin conocer los detalles de implementación, sólo la interfaz para comunicarse (parámetros)

Módulos (unidades, archivos .h, etc.)
Proporcionan la posibilidad de definir tipos y subprogramas, dividiéndolos en públicos y privados. La parte privada permite el ocultamiento de la información, el programador no tiene acceso a lo que no necesita.

Tipo de Dato Abstracto

Es un dato definido por el programador que se puede manipular igual que los tipos definidos por el lenguaje y comprende :

- Un conjunto de valores legales (pueden ser de tamaño indefinido)
- Un conjunto de operaciones que se pueden realizar sobre esos valores.

El usuario puede definir variables de estos tipos y operar con ellas.
Para construir un tipo de dato abstracto se debe poder:

- ✓ Describir la definición de un tipo
- ✓ Un conjunto de operadores
- ✓ Proteger los datos de forma que sólo puedan ser accedidos por rutinas proporcionadas para ese tipo
- ✓ Permitir instanciaciones del mismo (variables)

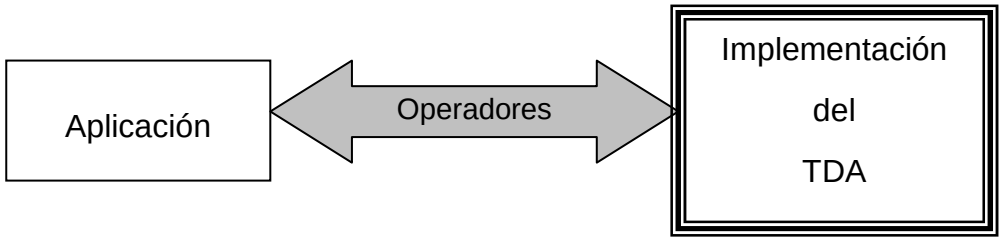
El TDA combina el tipo y sus operadores, los detalles de su implementación son transparentes al programador que lo utiliza sabiendo para qué sirve y cuándo usarlo y no cómo es y como lo hace.
Para ilustrar este concepto pensemos en el tipo real, lo utilizamos con +, -, * y /, sin saber cómo está representado en memoria (mantisa y exponente) y cómo opera internamente.
Sólo debe haber una visión lógica de un TDA, pero diferentes formas de implementarlo en cuanto a la representación en memoria y las funciones de acceso.

En la programación estructurada, los TDAs se implementan en módulos
Un módulo normalmente está compuesto por:
a. **Interfaz o protocolo de comunicación**, en lenguaje C, se refiere a los **archivos .h**:
   - ✓ *Contienen la declaración del nuevo tipo (**representación en memoria**) y*
   - ✓ *Los prototipos de las operaciones que estarán disponibles*

b. **Implementación**, en lenguaje C, se refiere a los **archivos .c**
   - ✓ *Contendrá el código fuente de cada operación*
   - ✓ *Cada implementación o "forma de llevarse a cabo" de **las operaciones**, estarán ocultas para el mundo exterior*

En resumen, un TDA bien modularizado a nivel de archivo, ocupará dos archivos, un .h y un .c, con el nombre del TDA.
El programa que utiliza el TDA, interactúa a través de la interfaz que le brindan las rutinas de acceso y mientras estas no cambien su **protocolo de comunicación**, el programa no se verá alterado. Aunque pueden cambiar la **representación en memoria** y los algoritmos plasmados en **las operaciones**.

Podemos pensar el TDA como un tipo de dato rodeado de una "muralla", que sólo pueden atravesar las rutinas de acceso (operadores) del TDA, que constituyen la interfaz o el protocolo de comunicación del TDA.. La separación del tipo, de los programas que lo utilizan (aplicación) se conoce como **encapsulamiento** y es una forma de **ocultación de la información**.



↳ **TDAMatriz**
- El arreglo bidimensional es un tipo de dato estructurado
- Permite agrupar elementos del mismo tipo, y acceder a cada uno de ellos mediante posición de fila y columna
- Lenguaje C sí nos permite trabajar con arreglos bidimensionales

¿Por qué definir un TDA matriz?

Para superar algunas limitaciones o condicionamientos:

- Los índices siempre comienzan desde 0 y eso no se puede cambiar.
- El tamaño del arreglo no es parte de la variable arreglo.
- Puedo intentar asignar valores a elementos fuera del rango del arreglo.
- El tamaño filas y columnas debe ser determinado al declarar la variable y si bien una matriz de 2x3 ocupa el mismo espacio que una de 3x2 (6 elementos) no puedo decidir la dimensión en tiempo de ejecución.
- Si es dinámica requiere malloc para crearla.

Si bien estas limitaciones se pueden superar con más programación, el código que debemos añadir hace más confusa la interpretación de las soluciones que utilizan la matriz.
Se deben rescatar los conceptos básicos, ¿qué almacena una matriz?, ¿cómo operar sus elementos? Y olvidarnos de los detalles de implementación.
Recordemos la abstracción: **quitar complejidad, dejar solo lo necesario.**

El objetivo es entonces definir un **TDAMatriz** que permita utilizar un arreglo bidimensional con menos limitaciones, escondiendo la complejidad. El tipo debe involucrar la dimensión lógica (Cantidad de filas y columnas) y el almacenamiento sobre un área de memoria lineal (estática o dinámica) que permita flexibilizar la dimensión (2x3 = 3x2). Las posiciones de fila y columna serán convertidas a un solo índice para el acceso individual de los elementos de la matriz. Si bien esto puede añadir complejidad al posicionamiento, otras operaciones sobre el total de elementos de la matriz, se simplificaran (búsquedas, promedios, máximos, mínimos, etc.)
Por ejemplo si se define:

```
int  m1[10][10], m2 [100];
```

en m1 se podrán almacenar matrices de a los sumo 10x10, mientras que en m2 podrán almacenarse matrices de 20x5, 25x4, 50x2 etc.

Todo detalle será escondido y sólo se interactúa con los operadores del TDA, a saber:

- Crea una matriz — `void crea (TDAmatriz *mat, int n, int m);`
- Obtiene la dimensión — `void dimension (TDAmatriz mat, int*n, int *m);`
- Lee una matriz — `void lee (TDAmatriz *mat);`
- Escribe una matriz — `void escribe (TDAmatriz mat);`
- Obtiene un elemento — `int obtienevalor (TDAmatriz mat, int i, int j);`
- Almacena un elemento — `void ponevalor (TDAmatriz *mat, int i, int j, int x);`
- Devuelve la Traspuesta — `TDAmatriz traspuesta (TDAmatriz mat);`
- Devuelve una fila
- Devuelve una columna
- Obtiene la posición de un valor
- Calcula el promedio
- Calcula Máximo y Mínimo — `void maxmin(TDAmatriz mat, int *max, int *min);`
- Verifica la existencia de un valor — `int esta(TDAmatriz mat, int x);`

```
/* archivo de interfaz  tdamatriz.h  - implementación estática*/
#define   DIM   100
typedef struct{
   int  n, m;
   int  elementos[DIM];
   }TDAmatriz;
/* operadores del TDA*/
void crea (TDAmatriz *mat, int n, int m);
void lee (TDAmatriz *mat);
void escribe (TDAmatriz  mat);
void dimension (TDAmatriz mat, int*n, int *m);
int obtienevalor (TDAmatriz  mat, int i, int j);
void ponevalor (TDAmatriz  *mat, int i, int j, int x);
int esta(TDAmatriz  mat, int x);
void maxmin(TDAmatriz  mat, int *max, int *min);
TDAmatriz traspuesta (TDAmatriz mat);
/*fin archivo de interfaz  tdamatriz.h */
```

/*programa que utiliza el TDAmatriz*/

```
#include “tdamatriz.h”
int main(void){
TDAmatriz mat, matri;
int i, j ,n, m,  max, min;
printf(“ingrese cantidad de filas y columnas de la matriz, restricción filas x columnas <=%d”, DIM);
scanf(“%d %d”, &n, &m);
crea(&mat, n, m);
lee(&mat);
escribe(mat);
dimension(mat, &n, &m);
printf(”primero  = %d   ultimo = %d \n”, obtienevalor(mat, 0,0), obtienevalor(mat,n-1, m-1));
if (esta(mat, 10))
     ponevalor(&mat, n/2, m/2, 0);
maxmin(mat, &max, &min);
printf(”maximo  = %d   minimo = %d \n”, max, min);
if(n = = m)
     {
     matri= traspuesta(mat);
     escribe(matri);
     }
else
     printf(“no es matriz cuadrada”);
return 0;
}
```

```
/*archivo de implementación tdamatriz.c - implementación estática  */
#include <stdio.h>
#include "tdamatriz.h"

void crea (TDAmatriz *mat, int n, int m)
    {
    mat->n=n;  mat->m=m;
    }

void lee (TDAmatriz *mat)
    {
    int i;
    printf("ingrese los elementos de la matriz por filas");
    for(i=0; i< mat->n *  mat->m; i++)
        scanf("%d", mat->elementos[i]);
    }

void escribe (TDAmatriz  mat)
    {
    int i, j;
    for(i=0; i< mat.n; i++)
        {
        for(j=0; j< mat.m; j++)
            printf("%d   ", mat.elementos[i * mat.m + j]);
        printf("\n");
        }
    }

void dimension (TDAmatriz mat, int*n, int *m)
    {
    *n= mat.n;  *m= mat.m;
    }

int obtienevalor (TDAmatriz  mat, int i, int j)
    {
    return  mat.elementos[i * mat.m + j];
    }

void ponevalor (TDAmatriz  *mat, int i, int j, int x)
    {
     mat->elementos[i * mat->m + j] = x;
    }

int maxmin(TDAmatriz  mat, int *max, int *min)
    {
    int i,   tope =  mat.n * mat.m;
    *max=mat.elementos[0];
    *min=mat.elementos[0];
    for(i=1; i< tope; i ++)
        if (*max < mat.elementos[i])
            *max= mat.elementos[i];
        else
            if (*min > mat.elementos[i])
                *min= mat.elementos[i];
    }
```

```
int esta(TDAmatriz  mat, int x);
     {
     int i=0, tope =  mat.n * mat.m;
     while (i < tope   &&   mat.elementos[i] != x)
          i++;
     return  i < tope ;
     }

TDAmatriz traspuesta (TDAmatriz mat)
     {
     int i, j;
     TDAmatriz  ma;
     if(mat.n = = mat.m)
          {
          for(i=0; i< mat.n; i++)
               for(j=0; j< mat.m; j++)
                    ma.elementos[j  * mat.n + i] = mat.elementos[i * mat.n + j];
          ma.n= mat.n ; ma.m= mat.n;
          }
     else
          {
          ma.n= 0 ; ma.m= 0;
          }
          return ma;
     }
/* fin archivo de implementación tdamatriz.c*/
```

Si **cambiamos la implementación de la matriz sobre un bloque dinámico de memoria**, cuyo tamaño se determina en función de las filas y columnas requeridas (en vez de un vector estático de 100 elementos). Se debe **modificar** en la interfaz **tdamatriz.h el tipo)** y en el desarrollo de **tdamatriz.c**. el cuerpo de los operadores **crea** y **traspuesta**
El programa que utiliza el TDAMatriz **no cambia.**

```
/* archivo de interfaz  tdamatrizd.h  - implementación dinámica*/
typedef struct{
   int  n, m;
   int  *pelem;  /*cambia arreglo de enteros por puntero a entero, no hay memoria reservada*/
   }TDAmatriz;
/* operadores del TDA, cambian las funciones crea y traspuesta */
void crea (TDAmatriz *mat, int n, int m);
…….
TDAmatriz traspuesta (TDAmatriz mat);
/*fin archivo de interfaz  tdamatrizd.h   */
```

```
/*archivo de implementación tdamatrizd.c  - implementación dinámica */
#include <stdio.h>
#include "tdamatrizd.h"
void crea (TDAmatriz *mat, int n, int m)
{
mat->n=n; mat->m=m;
mat->pelem= (int*)malloc(n*m*sizeof(int));  /* crea dinamicamente el espacio*/
}
……
TDAmatriz traspuesta (TDAmatriz mat)
    {
    int i, j;
   TDAmatriz  ma ;

   if(mat.n = = mat.m)
      {
      crea(&ma, mat.n, mat.n)   ; /*crea la matriz de retorno*/
      for(i=0; i< mat.n; i++)
         for(j=0; j< mat.n; j++)
            ma.pelem[j  * mat.n + i] = mat.pelem[i * mat.n + j];
      }
   else
      {
      ma.n= 0 ; ma.m= 0;
      }
    return ma;
   }
/* fin archivo de implementación tdamatrizd.c  */
```

PILA o STACK

Es una estructura de datos, que consiste en una "colección de elementos" todos del mismo tipo, donde las operaciones de insertar y eliminar un elemento se realizan por un extremo (tope). Sólo el elemento que está en el tope de la pila puede ser consultado o removido y si se agrega un elemento, éste se incorpora "sobre el tope o la cima" .
Un ejemplo típico es una pila de platos, donde ponemos sobre el último apilado y sacamos el que está en la cima (no se incorpora ni elimina por la base o el medio). También se conoce esta estructura con el nombre de LIFO (Last In First Out).
A pesar de su sencillez, es muy adecuada para la información que se comporta de este modo en el almacenamiento y recuperación. Es el caso de la *pila de ejecución* en la memoria principal, con espacio variable durante la ejecución del programa, crece hacia arriba y decrece hacia abajo. Cada subprograma invocado crea un registro de activación en la pila, allí se almacenan las variables locales, parámetros y dirección de retorno. Al finalizar el subrograma dicho espacio se libera. Cuando desde un subprograma A se invoca a otro subprograma B, los parámetros, variables locales y direcciones de retorno de B se apilan sobre las de A, quedando en el tope lo concerniente al subprograma activo. Cuando B finaliza, su información se desapila y queda en la cima el registro de A, que retoma la ejecución (suspendida por el llamado a B)

Para definir un Tipo de Dato Abstracto Pila, se debe esconder la implementación y definir un conjunto de operadores que permita declarar y utilizar variables del tipo.
A partir del tipo TPILA y sus operadores se podrán desarrollar algoritmos que almacenen y recuperen elementos de una Pila, solo es necesario conocer las cabeceras de los operadores, pero no los detalles de su implementación.

- Devuelve una Pila vacía — `void iniciap(TPILA *p);`
- Si la Pila no está llena, pone Elem en el tope de la misma, actualizándola — `void ponep(TPILA *p, TELEMENTO elem);`
- Si la Pila no está vacía, remueve el elemento del tope y lo devuelve en Elem — `void sacap(TPILA *p, TELEMENTO *pelem);`
- Consulta el elemento del Tope — `TELEMENTO consulta(TPILA p,TELEMENTO *pelem);`
- Indica si la Pila está vacía — `int vaciap(TPILA pila);`
- Indica si la pila está llena — `int llenap(TPILA pila);`

/* archivo de interfaz **tdapila.h - implementación estática***/

```
#define MAX   20
typedef int TELEMENTO;
typedef  struct{
    TELEMENTO vp[MAX] ;
    int tope;}TPILA;
void iniciap(TPILA *p);
int vaciap(TPILA pila);
int llenap(TPILA pila);
void ponep(TPILA *p, TELEMENTO elem);
void sacap(TPILA *p, TELEMENTO *pelem);
void consultap(TPILA pila, TELEMENTO *pelem);
```

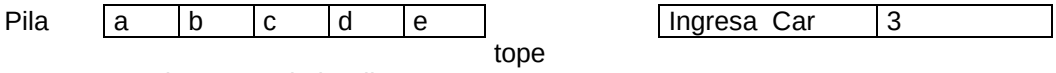
Ejemplo: leer un conjunto de números y mostrar los que son impares en orden inverso al que ingresaron.

```
#include <stdio.h>
#include "tdapila.h"
int main(){
    TPILA pila;
    TELEMENTO n;
    iniciap(&pila);
    while (scanf("%d", &n)==1)
      if(n%2)
         ponep(&pila, n);
    while(!vaciap(pila))
    {
        sacap(&pila,&n);
        printf("%d \n", n);
    }
    return 0;
}
```

Ejercicio 4 de la Práctica 4

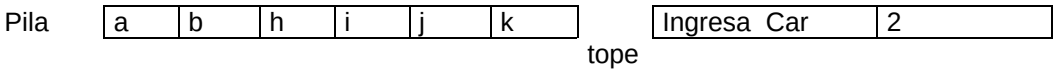
Ingresar un conjunto de caracteres y almacenar los que no son dígitos en una pila, cuando se trata de un dígito (´0´...´9´) sacar los últimos ingresados de acuerdo dígito o hasta que se vacíe la pila si no hubiera dicha cantidad en la misma. Ejemplo: a b c d e 3 h i j k 2 p

Los pone en la Pila

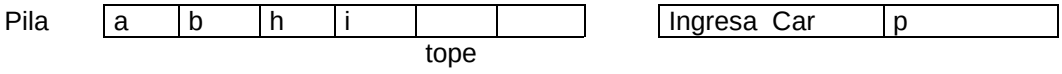


Saca 3 elementos de la Pila

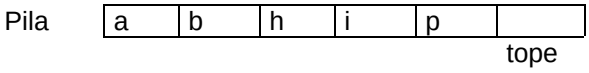
Los pone en la Pila



Saca 2 elementos de la Pila (*)



Lo pone en la Pila



(*) si en vez de un 2 ingresan caracteres que no son dígito, y la pila está llena hay que detener el proceso.

```
#include <stdio.h>
#include "tdapila.h"
int main()
{
 TPILA pila;
 TELEMENTO car, aux;
 int sigue=1;
 iniciap(&pila);
 while ((car=getchar())!=EOF && sigue)
      if('0'<=car && car<='9')
          for(  ; car >'0' && !vaciap(pila); sacap(&pila,&aux),car--);
      else
       if (!llenap(pila))
         ponep(&pila, car);
          else
         sigue=0;
 if(sigue)
   while(!vaciap(pila))
      {
        sacap(&pila,&car);
        putchar(car);
        }

 getchar();
 return 0;
}
```

COLA o QUEUE

Es una estructura de datos, que consiste en una colección de elementos, todos del mismo tipo, en la cual los elementos se incorporan por un extremo y se eliminan por otro.
Recibe también el nombre de FIFO (First In First Out), ya que el elemento que se almacenó primero es el primero en removerse. No se puede acceder a elementos intermedios. Un ejemplo muy común es la cola que se forma en la caja de un negocio o la ventanilla de un banco. El Sistema Operativo pone en cola a los procesos que requieren los distintos recursos del computador.
Para definir un Tipo de Dato Abstracto Cola, se debe esconder la implementación y definir un conjunto de operadores que permita declarar y utilizar variables del tipo.
A partir del tipo TCOLA y sus operadores se podrán desarrollar algoritmos que almacenen y recuperen elementos de una Cola, solo es necesario conocer las cabeceras de los operadores, pero no los detalles de su implementación.

```
/* archivo de interfaz  tdacola.h  - implementación estática*/
#define MAXCOLA   50
typedef int TELEMENTOC  ;
typedef struct{
   TELEMENTOC  items[MAXCOLA];
   int pri, ult;
}TCOLA;

void iniciac(TCOLA * c);
void ponec (TCOLA *c,  TELEMENTOC  elem);
void sacac(TCOLA *c,  TELEMENTOC  *e);
void consultac(TCOLA cola,  TELEMENTOC  *e);
int vaciac(TCOLA cola);
int llenac(TCOLA cola);
```

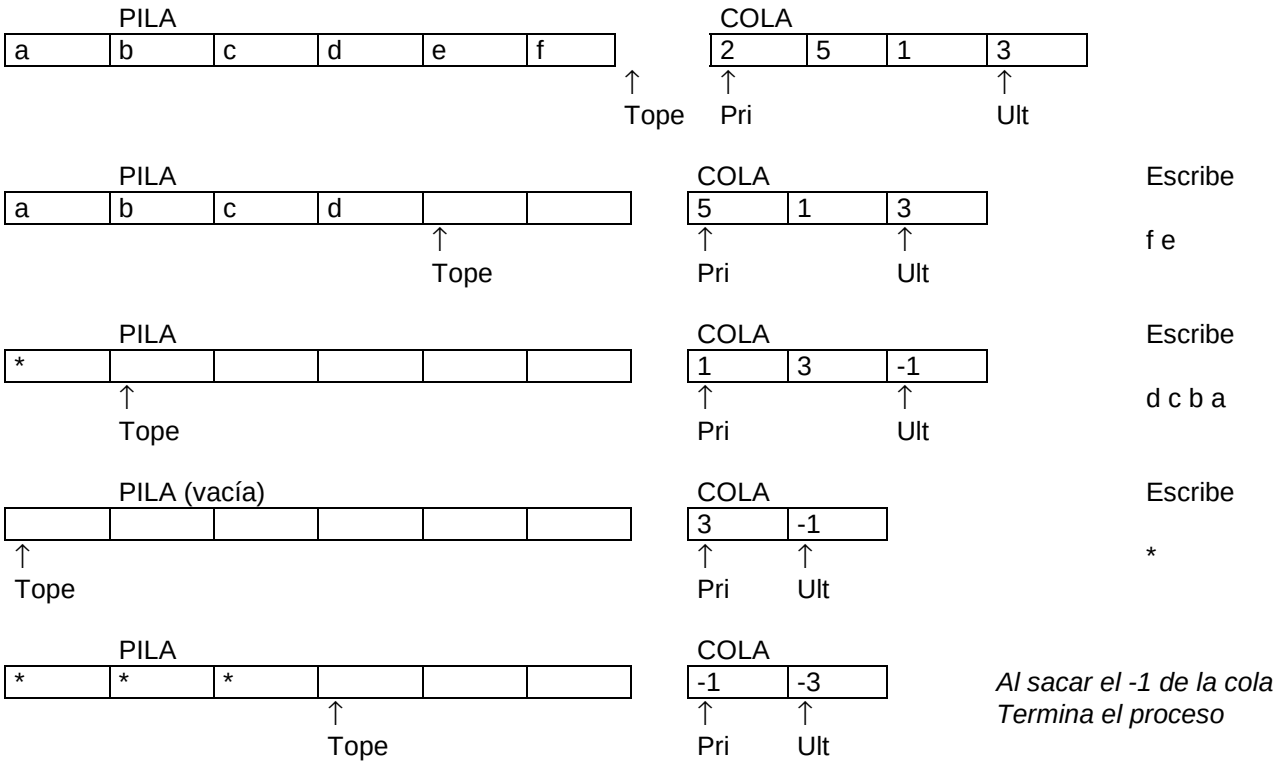
Ejemplo: leer un conjunto de números y mostrar los que son impares en el mismo orden al que ingresaron.

```
include<stdio.h>
#include"tdacola.h"
int main(void)
{
TCOLA  cola;
int n;
iniciac(&cola);
while (scanf("%d",&n))
    {
    if (n%2)
        ponec(&cola,n);
    }
while(!vaciac(cola))
    {
     sacac(&cola,&n);
     printf("%d",n);
    }
```

Ejercicio 3 de la Práctica 5

Dada una pila que contiene caracteres y una cola que contiene números naturales,
procesar cada elemento de la cola sacando de la pila tantos caracteres como el número indique. En caso de que la pila se vacíe antes de sacar la cantidad que el número determina, poner en ésta tantos * como caracteres faltantes resulten en la misma y almacenar en la cola esa diferencia negativa. El proceso termina cuando se hayan sacado todos los elementos de la cola original. (esto significa que quede vacía o que contenga los números negativos resultado del proceso de no encontrar en la pila la cantidad suficiente de caracteres)

Ejemplo




```
#include<stdio.h>
#include "tdacola.h"
#include "tdapila.h"
int main(void)
{
TCOLA  cola;
TPILA pila;
TELEMENTOC n, i,  negativo=0;
TELEMENTOP c;

iniciac(&cola); iniciap(&pila);

while (scanf("%d",&n))
    ponec(&cola,n);
fflush(stdin);
while((c=getchar())!=EOF)    //ingresar todos los caracteres seguidos
    ponep(&pila,c);          // y presionar Ctrl-Z+enter 2 veces.
```

```
while(!vaciac(cola) && !negativo)
    {
     sacac(&cola,&n);
     if(n < 0)
        negativo=1;
     else
        {
         while (!vaciap(pila) && n)
            {
            --n; sacap(&pila, &c); putchar(c);
            }
         if(n)
            {
             for(i=1; i<=n ; ponep(&pila,'*'), i++);
             ponec(&cola, -n);
            }
        }
    }
return 0;
}
```

IMPLEMENTACIONES ESTATICAS

/* archivo de implementación **tdapila.c - implementación estática** */

```
#include "tdapila.h"
void iniciap(TPILA *p)
{
  p->tope=0;
};
int vaciap(TPILA pila)
{
   return pila.tope==0;
};
int llenap(TPILA pila)
{
    return pila.tope==MAX;
};
void ponep(TPILA *p, TELEMENTO elem)
{
   p->vp[(p->tope)++]=elem;
};
void sacap(TPILA *p, TELEMENTO *pelem)
{
   --(p->tope);
   *pelem= p->vp[ p->tope];
};
void consultap(TPILA pila, TELEMENTO *pelem)
{
   *pelem= pila.vp[pila.tope -1];
};
```

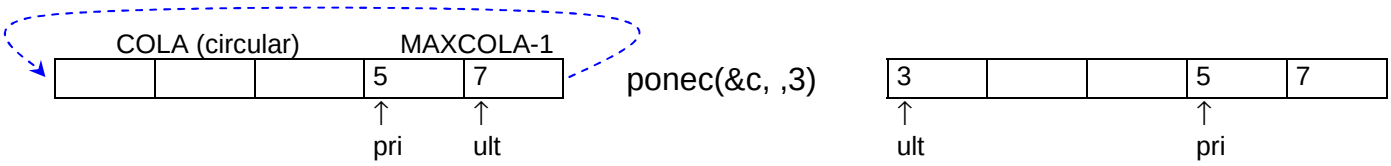
/* archivo de implementación **tdacola.c - implementación estática** */

```
#include"tdacola.h"
void iniciac (TCOLA * c)
  {
   c->pri=-1;  c->ult=-1;
  }
void ponec (TCOLA *c,  TELEMENTOC  elem)
   {
  if (c->ult != MAXCOLA -1)
      c->items[++(c->ult)] =elem;
  if (c->pri == -1)
      c->pri = 0;
   }
void sacac (TCOLA *c,  TELEMENTOC  *e)
   {
   if (c->pri != -1)
      {
       *e = c->items[c->pri];
       if (c->pri == c->ult)
          c->pri = c->ult = -1;
      else
```

```
            c->pri++;
        }
    }
```

```
void consultac (TCOLA cola,  TELEMENTOC  *e)
    {
    if (cola.pri!=-1)
        *e =cola.items[cola.pri];
    }
int vaciac(TCOLA cola)
    {
    return cola.pri== -1;
    }
int llenac(TCOLA cola)
   {
    return   cola.ult = = MAXCOLA -1;
   }
```

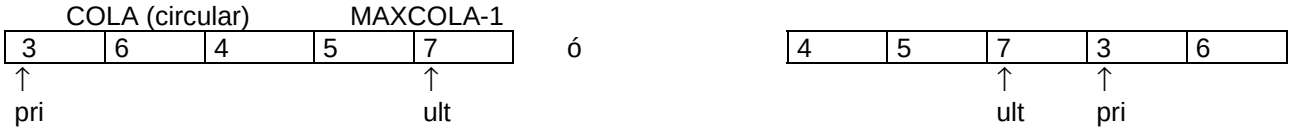
Otra posible implementación para la cola es almacenar los elementos en un arreglo y que el primer elemento se encuentre siempre almacenado en la primera posición del mismo. Con lo cual no es necesario incorporar, como parte del tipo, el campo pri (ya que siempre es 0).
Cada eliminación de un elemento significa un corrimiento hacia la izquierda de todos los elementos del arreglo, con lo cual se vuelve muy ineficiente. La inserción no cambia, ya que siempre se agrega al final.
Esta alternativa tiene como ventaja que en el arreglo no quedan lugares vacíos al comienzo, situación propia de la primera implementación cuando se efectúan eliminaciones sin llegar a vaciar la cola, ya que ult = MAXCOLA -1 puede ser verdadero lo que implica que la cola está llena, sin embargo existe espacio no aprovechado al comienzo.
Para solucionar esta situación sin un costo excesivo en cuanto al tiempo de proceso (corrimientos), se define la **Cola Circular**. Cuando se requiere almacenar un elemento y ult == MAXCOLA -1 pero hay espacio al comienzo, se considera 0 el siguiente de MAXCOLA -1, o sea se asigna 0 a ult y alli se coloca el elemento. Esto obliga a la redefinición de varios operadores (el tipo, todas las cabeceras y algunos operadores se mantienen igual).




```
void ponec (TCOLA *c,  TELEMENTOC  elem)
    {
     if (!llenac(*c))
         {
          If (c->ult == MAXCOLA -1)
                c->ult=0;
          else
               {
                c->ult++;
                if (c->pri == -1)
```

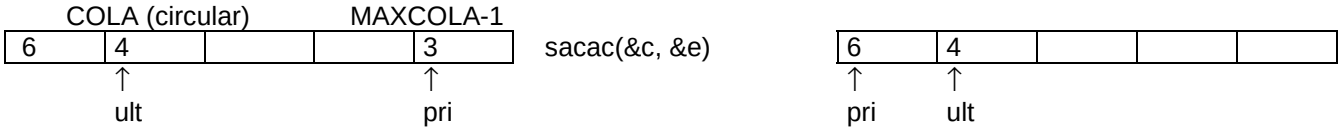
```
            c->pri =0;
          }
       c->items[c->ult] =elem;
       }
    }
```

La cola está llena cuando no hay espacio físico para insertar elementos




```
int llenac(TCOLA cola)
  {
   return   cola.pri==0 && cola.ult= = MAXCOLA -1 ||  cola.pri ==  cola.ult +1;
  }
```

Cuando saca, si pri (primero) está al final (MAXCOLA -1), debe pasar a la primera posición del arreglo




```
void sacac (TCOLA *c,  TELEMENTOC  *e)
   {
   if (c->pri!=-1)
      {
       *e =c->items[c->pri];
       if (c->pri == c->ult)
          c->pri= c->ult =-1;
      else
          if (c->pri == MAXCOLA -1)
               c->pri=0;
          else
               c->pri++;
       }
    }
```

IMPLEMENTACIONES DINAMICAS

```
/* archivo de interfaz  tdapila.h  - implementación dinámica */
typedef int TELEMENTOP;
typedef  struct nodo{
    TELEMENTOP dato ;
    struct nodo   *sig;}NODO;
typedef  NODO  *TPILA;
void iniciap(TPILA *p);
int vaciap(TPILA pila);
int llenap(TPILA pila);
void ponep(TPILA *p, TELEMENTOP elem);
void sacap(TPILA *p, TELEMENTOP *pelem);
void consultap(TPILA pila, TELEMENTOP *pelem);
```

```
/* archivo de implementación dinámica  tdapila.c */
#include <stdio.h>
#include "tdapilad.h"
void iniciap(TPILA *p)
    {
    *p=NULL;
    }
int vaciap(TPILA pila)
    {
     return pila==NULL;
    }
void ponep(TPILA *p, TELEMENTOP elem)
    {
     TPILA nuevo;
     nuevo= (TPILA)malloc(sizeof(NODO));
     nuevo->sig= *p; nuevo->dato=elem;
     *p=nuevo;
    }

void sacap(TPILA *p, TELEMENTOP *pelem)
    {
     TPILA aux;
     if(pila != NULL)
           {
            *pelem= (*p)->dato;    aux= *p;
            *p=(*p)->sig; free(aux);
           }
    }

void consultap(TPILA pila, TELEMENTOP *pelem)
    {
     if(pila != NULL)
        *pelem= pila->dato;
    }
```

```
/* archivo de interfaz  tdacola.h  - implementación dinámica */
typedef   int TELEMENTOC;
typedef  struct nodo{
   TELEMENTOC dato ;
   struct nodo   *sig;}NODO;
typedef struct cola{
   NODO *pri;
   NODO *ult;}TCOLA;
void iniciac (TCOLA * c);
void ponec (TCOLA *c,  TELEMENTOC  elem);
void sacac(TCOLA *c,  TELEMENTOC  *e);
void consultac(TCOLA cola,  TELEMENTOC  *e);
int vaciac(TCOLA cola);
/*int llenac(TCOLA cola); opcional para evaluar si hay memoria disponible en el heap*/


/* archivo de implementación  dinámica,  tdacola.h */
#include "tdacolad.h"
#include <stdio.h>
void iniciac (TCOLA * c){
   c->pri=c->ult=NULL;
   };
int vaciac(TCOLA cola){
   return cola.pri==NULL;
   }
void ponec (TCOLA *c,  TELEMENTOC  elem){
   NODO *pnuevo;
   pnuevo=(NODO*)malloc(sizeof(NODO));
   pnuevo->dato=elem;
   pnuevo->sig=NULL;
   if(c->pri==NULL)
      c->pri=pnuevo;
   else
      c->ult->sig=pnuevo;
   c->ult=pnuevo;
   }
void sacac(TCOLA *c,  TELEMENTOC  *e){
   NODO *paux;
   if(c->pri!=NULL){
      *e=c->pri->dato;
      paux=c->pri;
      c->pri=c->pri->sig;
      if(c->pri==NULL)
         c->ult=NULL;

      free(paux);
      }
   }
void consultac(TCOLA cola,  TELEMENTOC  *e){
   if(cola.pri!=NULL)
      *e=cola.pri->dato;
   }
```