Licenciatura en Sistemas
Programación de computadoras

Equipo docente: Jorge Golfieri, Natalia Romero, Romina Masilla y Nicolás Perez
Mails: jgolfieri@hotmail.com , nataliab_romero@yahoo.com.ar , romina.e.mansilla@gmail.com, nperez_dcao_smn@outlook.com
Facebook: https://www.facebook.com/groups/171510736842353
Git: ***http://***github.com/UNLASistemasProgramacion/Programacion-de-Computadoras

---

Unidad 7 - Extra:
Tipos de Datos Abstractos (TDA). Definición e implementación en C. Aplicaciones.

Bibliografía citada:
Luis Joyanes Aguilar, Andrés Castillo Sanz, y Lucas Sánchez García (2005) - C algoritmos, programación y estructuras de datos - McGraw-Hill España. Capítulo 17.

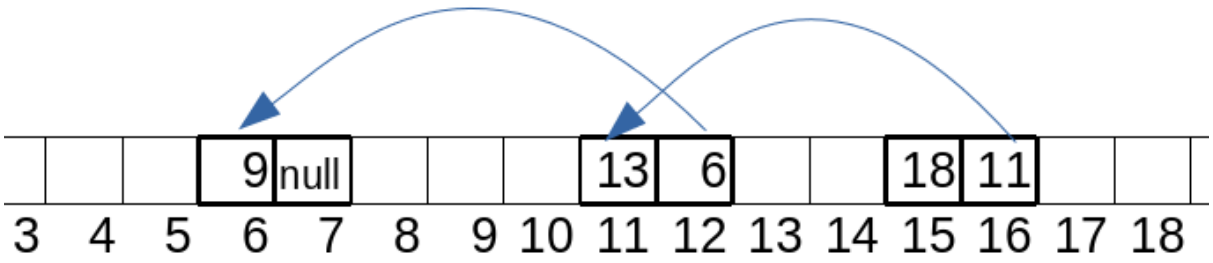
Pilas y colas

Las estructuras de datos que hemos empleado hasta ahora no podían aumentar su tamaño en el tiempo. Si utilizamos un array, por ejemplo, sus dimensiones deben conocerse de antemano, incluso en el caso de crearlo en memoria dinámica. Si nos ocurriera que necesitaramos ampliar la capacidad del mismo, deberíamos crear un array de mayor longitud y copiar los contenidos del original al mismo, para finalmente descartar al primero y seguir utilizando el segundo. Esto implica un overhead al agregar elementos, ya que en algún momento deberemos realizar esta operación de reserva de memoria y copia antes de poder agregar.
Hay problemas que requieren no sólo que las estructuras de datos que empleamos para resolverlos sean dinámicas (es decir, que puedan aumentar y disminuir de tamaño en el tiempo) sino que además exhiban un un comportamiento definido en cuanto a cómo se pueden agregar y eliminar elementos de las mismas. Las características del problema son las que determinan el comportamiento que debe exhibir la estructura de datos que lo representa. Así por ejemplo, si queremos modelar el comportamiento de la cola de la caja de un supermercado, necesitaremos una estructura de datos que pueda cambiar de tamaño (no siempre hay la misma cantidad de personas en la cola) sino que además tenga ciertas restricciones: Cuando queramos agregar un elemento sólo se podrá hacer por el final de la misma y cuando queramos remover, siempre lo haremos por el inicio. A continuación veremos dos estructuras de datos dinámicas básicas: las pilas y las colas. Utilizaramos TDA para representar las mismas.

Pila

Una pila (stack en inglés) es una estructura de datos dinámica en la que la restricción de agregado y remoción de elementos es que sólo puede realizarse por el mismo lado, la cabecera de la misma. Se trata de un estructura de tipo LIFO (Last In, First Out: El último que entra es el primero que sale), y podemos visualizarla como una pila de ropa planchada: para extraer la prenda que se encuentra en la posición 3 contando desde arriba, deberemos retirar las dos que se encuentran encima previamente, de otro modo se desarmaría la pila.
Las operaciones primitivas de una pila son entonces: apilar (push en inglés) que agrega un elemento encima del que se encuentre en la cima de la pila o inicia la pila si no hay ninguno, desapilar (pop en inglés) que quita y nos devuelve el primer elemento de la pila, alguna operación que nos informe si ya no quedan elementos en la pila (isEmpty en inglés) y otra para conocer la longitud (la cantidad de elementos, length en inglés) de la misma.

A fin de que la estructura de datos pueda crecer, emplearemos la siguiente estrategia: Cada vez que queramos agregar un elemento a la misma, reservaremos memoria para el elemento que debemos apilar y para un puntero que nos indicará cuál es el elemento siguiente. Veámoslo con un ejemplo, supongamos que queremos apilar los enteros 9, 13 y 18:



Cada vez que apilamos un elemento, reservamos memoria para el mismo y para un puntero al siguiente. El primer elemento apilado, 9, está seguido por null, ya que no tiene elemento siguiente. El puntero del segundo elemento (13), apunta a la memoria reservada para el primer elemento, y el del tercero a la del segundo. Esto nos permite reservar memoria a medida que la necesitamos y aún así mantener, por medio de los punteros al siguiente, el orden de los elementos en la pila. Manteniendo un puntero al último elemento apilado, a través del mismo podemos llegar al anterior, y así sucesivamente, hasta llegar a aquel cuyo puntero al siguiente contiene null, lo que nos indica que no hay más elementos luego del mismo. El puntero a la cabecera de la pila vale 15.
Podemos ver que cada conjunto (dato + puntero al siguiente) conforma en sí mismo una estructura de datos. Esta estructura se denomina nodo y resulta ideal representarla por medio de un TDA, ya que necesitaremos operaciones para crearlo, destruirlo, para establecer y obtener datos y punteros.
Daremos un paso más y, a fin de que en el nodo se pueda almacenar cualquier tipo de datos (no sólo un entero), definiremos que en lugar del dato almacenaremos un puntero al dato, pero como no sabemos de antemano de qué tipo de datos puede tratarse, lo representaremos con un puntero a void (void*). De este modo un nodo podrá almacenar una referencia (es decir un puntero) a un entero, una array, una estructura, otro TDA, es decir, cualquier otro tipo de datos.

Veamos la implementación de nodo por medio de un TDA. Primero la interfaz, Nodo.h:

```
#ifndef NODO_H_INCLUDED
#define NODO_H_INCLUDED

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

typedef void* PtrDato;

/* Tipo de Estructura de los Nodos de la Lista. */
typedef struct Nodo {
      PtrDato dato; // dato almacenado
      struct Nodo *sgte; // puntero al siguiente
} Nodo;

// puntero a nodo
typedef Nodo* PtrNodo;

// Terminador de nodo. En Nodo.c se asigna NULL
const void* SinNodoSgte;

// Operación de construccion (constructor)
// Precondicion: El nodo no debe haberse creado
// Postcondición: Se crea el nodo con null en PtrDato y SinNodoSgte en
sgte
// Parámetros:
```

```
//  dato: Puntero al dato a almacenar
// Devuelve puntero al nodo creado
PtrNodo crearNodo(PtrDato dato);

// Operación de destruccion (destructor)
// Precondicion: El nodo debe haberse creado
// Postcondición: Se Libera la memoria del nodo eliminado
// Parámetros:
//  puntero al nodo a eliminar
// Devuelve NULL
PtrNodo destruirNodo(PtrNodo nodo);

// Operación de establecimiento de datos
// Precondicion: nodo creado con crearNodo()
// Postcondición: almacena el puntero al dato proporcionado en dato
//                en el nodo apuntado por ptrNodo.
// Parámetros:
//  ptrNodo: puntero al nodo
//  PtrDato: Puntero al dato a almacenar
// No devuelve valor
void setDato(PtrNodo nodo, PtrDato dato);

// Operación de obtención de datos
// Precondicion: nodo creado con crearNodo()
// Postcondición: obtiene el dato almacenado en el nodo apuntado por
ptrNodo.
// Parámetros:
//  ptrNodo: puntero al nodo
// Devuelve puntero al dato almacenado
PtrDato getDato(PtrNodo nodo);

// Operación de establecimiento del siguiente nodo
// Precondicion: esteNodo creado con crearNodo()
// Postcondición: se establece el nodo siguiente al actual
// Parámetros:
//  esteNodo: puntero al nodo
// No devuelve valor
void setSiguiente(PtrNodo esteNodo, PtrNodo siguienteNodo);

// Operación de obtención del siguiente nodo
// Precondicion: esteNodo creado con crearNodo()
// Postcondición: se obtiene el nodo siguiente al actual
// Parámetros:
//  nodo: puntero al nodo
// No devuelve valor
PtrNodo getSiguiente(PtrNodo nodo);
```

```
#endif // NODO_H_INCLUDED
```

Y luego la implementación propiamente dicha, Nodo.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "Nodo.h"
#include "Util.h"

const void* SinNodoSgte=NULL;

PtrNodo crearNodo(PtrDato dato){
	PtrNodo nodo=(PtrNodo)obtenerMemoria(sizeof(Nodo));
	nodo->dato=dato;
	nodo->sgte=SinNodoSgte;
	return nodo;
}

// Operación de destruccion (desstructor)
// Precondicion: El nodo debe haberse creado
// Postcondición: Se Libera la memoria del nodo eliminado
// Parámetros:
//  puntero al nodo a eliminar
// Devuelve NULL
PtrNodo destruirNodo(PtrNodo nodo){
	// la destrucción del dato queda por cuenta del usuario
	free(nodo);
	return SinNodoSgte;
}


// Operación de establecimiento de datos
// Precondicion: nodo creado con crearNodo()
// Postcondición: almacena el dato proporcionado en el nodo apuntado por
ptrNodo.
// Parámetros:
//  ptrNodo: puntero al nodo
//  PtrDato: Puntero al dato a almacenar
// No devuelve valor
void setDato(PtrNodo nodo, PtrDato dato){
	nodo->dato=dato;
}

// Operación de obtención de datos
// Precondicion: nodo creado con crearNodo()
```

```
// Postcondición: obtiene el dato almacenado en el nodo apuntado por
ptrNodo.
// Parámetros:
//  ptrNodo: puntero al nodo
// Devuelve puntero al dato almacenado
PtrDato getDato(PtrNodo nodo){
      return nodo->dato;
}

// Operación de establecimiento del siguiente nodo
// Precondicion: esteNodo creado con crearNodo()
// Postcondición: se establece el nodo siguiente al actual
// Parámetros:
//  esteNodo: puntero al nodo
// No devuelve valor
void setSiguiente(PtrNodo esteNodo, PtrNodo siguienteNodo){
      esteNodo->sgte=siguienteNodo;
}

// Operación de obtención del siguiente nodo
// Precondicion: esteNodo creado con crearNodo()
// Postcondición: se obtiene el nodo siguiente al actual
// Parámetros:
//  nodo: puntero al nodo
// No devuelve valor
PtrNodo getSiguiente(PtrNodo nodo){
      return nodo->sgte;
}
```

Una vez definido el TDA para el nodo, necesitamos definir la pila propiamente dicha. Lo haremos por medio de otro TDA, que utilizará el nodo recién definido. Primero la interfaz, Pila.h:

```
#ifndef PILA_H_INCLUDED
#define PILA_H_INCLUDED

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "Nodo.h"

typedef struct Pila{
      PtrNodo primero;        // puntero al primer nodo de la pila
} Pila;

typedef Pila* PtrPila;
```

```
// Terminador de pila. En Pila.c se define como NULL;
const void* FinPila;

/*************************************************************************
*****/
/* Definicion de Primitivas */
/*--------------------------*/

// Operación de construccion (constructor)
// Precondicion: la pila no debe haber sido creada.
// Postcondición: píla queda creada vacía y preparada para ser usada.
// Parámetros:
//  Ninguno
// Devuelve puntero a la pila creada
PtrPila crearPila();

// Operación de destruccion (destructor)
// Precondicion: la pila debe haber sido creada.
// Postcondición: la pila queda vacía y la memoria que ocupaba se libera
// Parámetros:
//  Ninguna
// Devuelve NULL
PtrPila destruirPila(PtrPila pila);

// Operación de verificación de estado
// Precondicion: la pila debe haber sido creada.
// Postcondición: ninguna
// Parámetros:
//  pila: Puntero a la pila que se desea saber si se encuentra vacía
// Devuelve true si la pila se encuentra vacía (primero=FinPila), false de
otro modo
bool pilaVacia(PtrPila pila);

// Operación de obtención de la longitud de la pila
// Precondicion: la pila debe haber sido creada.
// Postcondición: Se obtiene la longitud de la pila (si vacía =0)
// Parámetros:
//  pila: puntero a la pila
// Devuelve longitud de la pila
int longitudPila(PtrPila pila);

// Operación de agregado a pila
// Precondicion: la pila debe haber sido creada.
// Postcondición: Se inserta el dato antes de la primra posición
// Parámetros:
//  pila: puntero a la pila
//  dato: puntero al dato a insertar
```

```
// Devuelve puntero al nodo en que se insertó el dato
void apilar(PtrPila pila, PtrDato dato);

// Operación de agregado antes de la cabecera
// Precondicion: la pila debe haber sido creada.
// Postcondición: Se inserta el dato antes de la cabecera de la pila
// Parámetros:
//  pila: puntero a la pila
//  dato: puntero al dato a insertar
// Devuelve puntero al nodo en que se insertó el dato
PtrDato desapilar(PtrPila pila);

// Operación de agregado al final de la pila
// Precondicion: la pila debe haber sido creada.
// Postcondición: Se agrega el dato luego del ultimo nodo
// Parámetros:
//  pila: puntero a la pila
//  dato: puntero al dato a insertar
// Devuelve puntero al nodo en que se insertó el dato
PtrDato primeroPila(PtrPila pila);

#endif // PILA_H_INCLUDED
```

Luego la implementación, Pila.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "Pila.h"
#include "Util.h"

const void* FinPila=NULL;

PtrPila crearPila(){
      PtrPila pila=(PtrPila)obtenerMemoria(sizeof(Pila));
      pila->primero=FinPila;
      return pila;
}

PtrPila destruirPila(PtrPila pila){
      // desapilamos cada uno de los nodos y para eliminamos.  Si haba
datos a eliminar,
      // eso es responsabilidad del usuario
      while(!pilaVacia(pila)){
      desapilar(pila);
      }

      free(pila);
```

```
	return NULL;
}

bool pilaVacia(PtrPila pila){
	return (pila->primero==FinPila);
}

int longitudPila(PtrPila pila){
	// la longitud es cuantos nodos hay en la pila. Para saberlo, hay
que recorrerla
	int longitud=0;
	PtrNodo nodo=pila->primero;
	while(nodo!=FinPila){
	nodo=getSiguiente(nodo);
	longitud++;
	}
	return longitud;
}

// Otra versión, recursiva
int longitudPilaRecur(PtrPila pila){
	int longitud=longitudPilaAyudante(pila->primero,0);
	return longitud;
}

int longitudPilaAyudante(PtrNodo nodo, int longitud){
	// la longitud es cuantos nodos hay en la pila. Para saberlo, hay
que recorrerla

	if(nodo==FinPila){
	return longitud;
	} else {
	longitudPilaAyudante(getSiguiente(nodo), longitud++);
	}
}

void apilar(PtrPila pila, PtrDato dato){
	// creamos un nodo, le asignamos el dato y lo ponemos al frente de
la pila
	if(pila!=NULL){
	PtrNodo nodo=crearNodo(dato);
	setDato(nodo,dato);
	setSiguiente(nodo,pila->primero);
	pila->primero=nodo;
	}
}
```

```
PtrDato desapilar(PtrPila pila){
	PtrDato dato=NULL;
	if(!pilaVacia(pila)){
	PtrNodo nodo=pila->primero;
	pila->primero=getSiguiente(nodo);
	dato=getDato(nodo);
	nodo=destruirNodo(nodo);
	}
	return dato;
}

PtrDato primeroPila(PtrPila pila){
	// devuelve el dato de la primera posición de la pila
	// pero sin desapilarlo
	PtrDato dato=NULL;
	if(!pilaVacia(pila)){
	PtrNodo nodo=pila->primero;
	dato=getDato(nodo);
	}
	return dato;
}
```

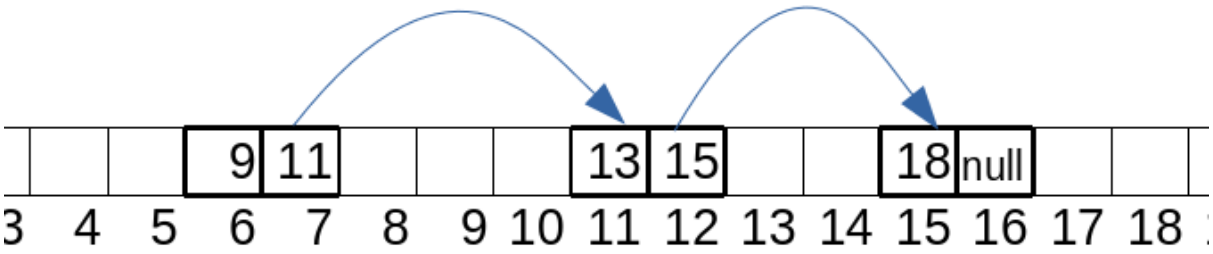
Y finalmente, un ejemplo de utilización (main.c):

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

int main()
{
	PtrPila pila=crearPila();

	//apilamos strings en memoria automática. Qué pasaría si lo
	// hiciéramos en una función que devuelve la pila?

	// En qué orden salen los elementos al desapilar?

	apilar(pila,"Primero");
	apilar(pila,"Segundo");
	apilar(pila,"Tercero");
	apilar(pila,"Cuarto");

	printf("Longitud de la pila: %d\n",longitudPila(pila));

	// desapilamos y mostramos un elemento
	printf("%s\n",(char*)desapilar(pila));
```

```
        printf("Longitud de la pila: %d\n",longitudPila(pila));

        // desapilamos y mostramos al resto

        while(!pilaVacia(pila)){
        printf("%s\n",(char*)desapilar(pila));
        }

        printf("Longitud de la pila: %d\n",longitudPila(pila));

        // ahora apilamos strings en memoria dinámica. stringDinamico
        // está definida en Util.h

        apilar(pila,stringDinamico("esta"));
        apilar(pila,stringDinamico("es"));
        apilar(pila,stringDinamico("una"));
        apilar(pila,stringDinamico("prueba"));

        // desapilamos, mostramos y reclamamos memoria.
        while(!pilaVacia(pila)){
        char* string=(char*)desapilar(pila);
        printf("%s\n",string);
        free(string); // Si no hacemos esto? Qué pasa?
        }

        // forzamos enteros sin punteros. Cuidado con el tamaño máximo
        apilar(pila,1);
        apilar(pila,2);
        apilar(pila,3);
        apilar(pila,-4);

        while(!pilaVacia(pila)){
        printf("%d\n",(int)desapilar(pila));
        }

        printf("\n%d\n",(int)desapilar(pila));

        // finalmente destruimos la pila.
        pila=destruirPila(pila);

        return 0;
}
```

Cola

La implementación de una cola (queue en inglés) es muy similar a la de una pila, con la salvedad de que el mecanismo de agregado y extracción de elementos cambia: en una cola sólo pueden agregarse elementos por el extremo opuesto al que se los extrae. Se trata de una estructura de datos de tipo FIFO (First In, First Out, el que entra primero, sale primero) y sigue la lógica de la cola del supermercado: Siempre se atiende al primero de la fila (se extrae por la cabecera) y cada nuevo comprador que llega se pone el el último lugar (se agrega por el final). En el ejemplo siguiente, encolamos los enteros 9, 13 y 18 en ese orden.



Las operaciones primitivas de una cola son entonces: encolar (enqueue en inglés) que agrega un elemento luego del último de la cola o la inicia si no hay ninguno; desencolar (dequeue en inglés) que quita y nos devuelve el primer elemento de la cola, estableciendo al siguiente como el primero; alguna operación que nos informe si ya no quedan elementos en la cola (isEmpty en inglés) y otra para conocer la longitud (la cantidad de elementos, length en inglés) de la misma.
Utilizaremos la misma estrategia que con la pila, es decir, utilizaremos nodos para conformar la cola. Primera ventaja de haber definido a nodo como un TDA: ya lo tenemos implementado. A diferencia de la pila, no nos alcanza con que la cola mantenga un puntero al nodo cabecera de la misma: ahora es necesario además mantener otro puntero al nodo final. De este modo, podremos agregar al final de la cola sin necesidad de recorrerla desde la primera posición para encontrar el último nodo. En el ejemplo anterior, el puntero a la cabecera vale 6 y el que apunta al final, 15.

Veamos entonces la implementación de una cola con TDA. Primero la interfaz, Cola.h:

```
#ifndef COLA_H_INCLUDED
#define COLA_H_INCLUDED

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "Nodo.h"

typedef struct Cola{
      PtrNodo primero;        // puntero al primer nodo de la cola
      PtrNodo ultimo;   // puntero al ultimo nodo de la cola

} Cola;

typedef Cola* PtrCola;
```

```
// Terminador de Cola. En Cola.c se define como NULL;
const void* FinCola;

/*************************************************************************
*****/
/* Definicion de Primitivas */
/*--------------------------*/

// Operación de construccion (constructor)
// Precondicion: la cola no debe haber sido creada.
// Postcondición: cola queda creada vacía y preparada para ser usada.
// Parámetros:
//  Ninguno
// Devuelve puntero a la cola creada
PtrCola crearCola();

// Operación de destruccion (destructor)
// Precondicion: la cola debe haber sido creada.
// Postcondición: la cola queda vacía y la memoria que ocupaba se libera
// Parámetros:
//  Ninguna
// Devuelve NULL
PtrCola destruirCola(PtrCola cola);

// Operación de verificación de estado
// Precondicion: la cola debe haber sido creada.
// Postcondición: ninguna
// Parámetros:
//  cola: Puntero a la cola que se desea saber si se encuentra vacía
// Devuelve true si la cola se encuentra vacía (primero=FinCola), false de
otro modo
bool colaVacia(PtrCola cola);

// Operación de obtención de la longitud de la cola
// Precondicion: la cola debe haber sido creada.
// Postcondición: Se obtiene la longitud de la cola (si vacía =0)
// Parámetros:
//  cola: puntero a la cola
// Devuelve longitud de la cola
int longitudCola(PtrCola cola);

// Operación de agregado a cola
// Precondicion: la cola debe haber sido creada.
// Postcondición: Se inserta el dato luego de la última posición
// Parámetros:
//  cola: puntero a la cola
```

```
//  dato: puntero al dato a insertar
// Devuelve puntero al nodo en que se insertó el dato
void encolar(PtrCola cola, PtrDato dato);

// Operación de agregado antes de la cabecera
// Precondicion: la cola debe haber sido creada.
// Postcondición: Se inserta el dato antes de la cabecera de la cola
// Parámetros:
//  cola: puntero a la cola
//  dato: puntero al dato a insertar
// Devuelve puntero al nodo en que se insertó el dato
PtrDato desencolar(PtrCola cola);

// Operación de obtención del primero de la cola
// Precondicion: la cola debe haber sido creada.
// Postcondición: Se obtiene el dato del primer nodo sin desenconlarlo
// Parámetros:
//  cola: puntero a la cola
// Devuelve puntero al dato de la primera posición
PtrDato primeroCola(PtrCola cola);

#endif // COLA_H_INCLUDED
```

Luego la implementación propiamente dicha, Cola.c:

```
#include <stdbool.h>
#include "Cola.h"
#include "Util.h"

const void* FinCola=NULL;

PtrCola crearCola(){
      PtrCola cola=(PtrCola)obtenerMemoria(sizeof(Cola));
      cola->primero=FinCola;
      cola->ultimo=FinCola;
      return cola;
}

PtrCola destruirCola(PtrCola cola){
      // desencolamos cada uno de los nodos y los eliminamos.  Si había
      //  datos a eliminar, eso es responsabilidad del usuario
      while(!colaVacia(cola)){
      desencolar(cola);
      }
      free(cola);
      return NULL;
}
```

```
bool colaVacia(PtrCola cola){
	return (cola->primero==FinCola);
}

int longitudCola(PtrCola cola){
	// la longitud es cuantos nodos hay en la pila. Para saberlo, hay
que recorrerla
	int longitud=0;
	PtrNodo nodo=cola->primero;
	while(nodo!=FinCola){
	nodo=getSiguiente(nodo);
	longitud++;
	}
	return longitud;
}

void encolar(PtrCola cola, PtrDato dato){
	if(cola!=NULL){
	PtrNodo nodo=crearNodo(dato);
	setDato(nodo,dato);
	if(colaVacia(cola)){
		cola->primero=nodo;
		cola->ultimo=nodo;
	} else {
		setSiguiente(cola->ultimo,nodo);
		cola->ultimo=nodo;
	}
	}
}

PtrDato desencolar(PtrCola cola){
   PtrDato dato=NULL;
	if(!colaVacia(cola)){
	PtrNodo nodo=cola->primero;
	cola->primero=getSiguiente(nodo);
	dato=getDato(nodo);
	if(cola->primero==FinCola){
		cola->ultimo=FinCola;
	}
	nodo=destruirNodo(nodo);
	}
	return dato;
}


PtrDato primeroCola(PtrCola cola){
```

```
   // devuelve el dato de la primera posición de la cola
      // pero sin desencolarlo
      PtrDato dato=NULL;
      if(!colaVacia(cola)){
      PtrNodo nodo=cola->primero;
      dato=getDato(nodo);
      }
      return dato;
}
```

Y finalmente, un ejemplo de utilización (main.c):

```
#include <stdio.h>
#include <stdlib.h>
#include "Cola.h"
#include "Util.h"

int main()
{
      PtrCola cola=crearCola();

      encolar(cola,"Primero");
      encolar(cola,"Segundo");
      encolar(cola,"Tercero");
      encolar(cola,"Cuarto");

      printf("Longitud de la cola: %d\n",longitudCola(cola));

      // desencolamos al primero
      // Qué pasa con el orden?
      printf("%s\n",(char*)desencolar(cola));

      printf("Longitud de la cola: %d\n",longitudCola(cola));

      // desencolamos y mostramos el resto
      while(!colaVacia(cola)){
      printf("%s\n",(char*)desencolar(cola));
      }

      printf("Longitud de la cola: %d\n",longitudCola(cola));

      encolar(cola,stringDinamico("esta"));
      encolar(cola,stringDinamico("es"));
      encolar(cola,stringDinamico("una"));
      encolar(cola,stringDinamico("prueba"));

      // desencolamos, mostramos y reclamamos memoria.
```

```
        while(!colaVacia(cola)){
        char* string=(char*)desencolar(cola);
        printf("%s\n",string);
        free(string);
        }

        // Forzamos enteros
        encolar(cola,1);
        encolar(cola,2);
        encolar(cola,3);
        encolar(cola,-4);

        while(!colaVacia(cola)){
        printf("%d\n",(int)desencolar(cola));
        }

        // Finalmente destruimos
        cola=destruirCola(cola);

        return 0;
}
```