

Lista enlazada, pila y cola

—

Ejercicio 1

Nelson logró abrir su negocio para arreglar y mejorar bicicletas. Antes que nada, juntó los repuestos que podría llegar a necesitar para trabajar. Tiene algunos viejos y otros los compró nuevos.

Una vez que consiguió todos, los ordenó por tamaño y los guardó en el siguiente TDA caja_repuestos_t:

```
#ifndef CAJA_REPUESTOS_H
#define CAJA_REPUESTOS_H

#include <stdbool.h>

typedef struct repuesto {
    char nombre[MAX_NOMBRE];
    int codigo;
    int tamano;        // en mm
    bool oxidado;
    int id_repuesto;
} repuesto_t;

typedef struct caja_repuestos caja_repuestos_t;

/*
 * Pre: -
 * Post: Crea una caja de repuestos vacía.
 */
caja_repuestos_t* caja_repuestos_crear();

/*
 * Pre: caja válida.
 * Post: Libera toda la memoria utilizada.
 */
void caja_repuestos_destruir(caja_repuestos_t* caja);

/*
 * Pre: caja válida.
 * Post: Apila un repuesto.
 */
void caja_repuestos_apilar(caja_repuestos_t* caja, repuesto_t repuesto);
```

```

/*
 * Pre: caja válida y no vacía.
 * Post: Desapila el repuesto del tope y devuelve un
 * puntero reservado dinámicamente.
 */
repuesto_t* caja_repuestos_desapilar(caja_repuestos_t* caja);

/*
 * Pre: caja válida.
 * Post: Devuelve el repuesto del tope o NULL.
 */
repuesto_t* caja_repuestos_tope(caja_repuestos_t* caja);

/*
 * Pre: caja válida.
 * Post: Devuelve true si está vacía.
 */
bool caja_repuestos_vacia(caja_repuestos_t* caja);

#endif

```

Se pide:

- Dado el TDA `caja_de_repuestos_t` cargado, crear un procedimiento que revise cada repuesto y tire aquellos que no pueda utilizar sabiendo que no le sirven:
 - los tornillos que tengan código mayor a 13000,
 - y tampoco aquellos repuestos que tengan más de 50 mm y estén oxidados.

Ejercicio 2

Una vez que Nelson pudo organizar sus repuestos, abrió las puertas de su negocio y por suerte tiene muchísimos pedidos de clientes que recibe por mail.

Para atenderlos a todos como se merecen, los organiza en una lista enlazada:

```

#ifndef MAILS_PEDIDOS_H
#define MAILS_PEDIDOS_H

```

```

#include <stdbool.h>

#define MAX_MAIL 100

// Nodo de la lista enlazada (pedido recibido por mail)
typedef struct pedido_mail {
    int id_bicicleta;
    int id_repuesto;
    char mail_cliente[MAX_MAIL];
    bool es_spam;

    struct pedido_mail* siguiente;
} pedido_mail_t;

typedef struct bandeja bandeja_t;

/**
 * Pre: -
 * Post: Devuelve una bandeja vacía reservada en el heap,
 *        o NULL si la reserva falla.
 *        La bandeja devuelta debe liberarse con bandeja_destruir().
 */
bandeja_t* bandeja_crear();

/**
 * Pre: bandeja válida (puede estar vacía).
 * Post: Libera la bandeja y TODOS los pedidos que sigan dentro de ella.
 *        Luego de esta llamada el puntero 'bandeja' queda invalidado y no
 *        debe usarse. No libera los pedidos que el usuario haya sacado
 *        previamente con bandeja_quitar_primer(): esos son responsabilidad
 *        de quien los quitó.
 */
void bandeja_destruir(bandeja_t* bandeja);

/**
 * Pre: bandeja válida.
 * Post: Crea un nuevo pedido (copiando los datos del struct recibido,
 *        ignorando su campo 'siguiente') y lo agrega al comienzo. El pedido

```

```

*      queda en posesión de la bandeja: NO debe liberarse por separado
*      mientras siga dentro de ella.
*/
void bandeja_agregar_al_inicio(bandeja_t* bandeja, pedido_mail_t pedido);

/**
* Pre: bandeja válida.
* Post: Crea un nuevo pedido (copiando los datos del struct recibido,
*       ignorando su campo 'siguiente') y lo agrega al final. El pedido
*       queda en posesión de la bandeja: NO debe liberarse por separado
*       mientras siga dentro de ella.
*/
void bandeja_agregar_al_final(bandeja_t* bandeja, pedido_mail_t pedido);

/**
* Pre: bandeja válida y no vacía.
* Post: Saca el primer pedido de la lista y lo devuelve.
*       La bandeja deja de ser dueña del pedido: a partir de acá la
*       memoria del pedido pasa a ser responsabilidad de quien llama, que
*       DEBE liberarlo con free() cuando termine de usarlo. El campo
*       'siguiente' del pedido devuelto queda en NULL.
*/
pedido_mail_t* bandeja_quitar_primerero(bandeja_t* bandeja);

/**
* Pre: bandeja válida.
* Post: Devuelve un puntero al primer pedido sin sacarlo de la lista, o
*       NULL si la bandeja está vacía. El pedido sigue siendo propiedad de
*       la bandeja: NO debe liberarse ni modificarse su campo 'siguiente'.
*/
pedido_mail_t* bandeja_primerero(bandeja_t* bandeja);

/**
* Pre: mail_cliente y mensaje son strings válidos.
*
* Post:
*
* Simula el envío de una respuesta al cliente. No reserva ni libera
* memoria: solo lee las cadenas recibidas (no toma posesión de ellas).
*/
void bandeja_responder_cliente(char* mail_cliente, char* mensaje);

```

```
#endif
```

Un pedido NO es válido cuando el mail es spam o cuando el id del repuesto es inválido (menor a 0).

- Realizar una función que reciba la lista de pedidos y los procese de manera que descarte aquellos pedidos que no sean válidos y conteste los mails de los que se puedan tomar. El mensaje tiene que ser "Su bicicleta será atendida por Nelson a la brevedad." para todos.

Ejercicio 3

Nelson ya tiene sus repuestos organizados y procesó todos los pedidos válidos de su bandeja. Ahora necesita ponerse a trabajar.

Para organizarse, va a cargar una cola de bicicletas a reparar (cola_bicicletas_t) y luego las irá atendiendo de a una, buscando el repuesto necesario en su caja.

El struct de bicicleta es el siguiente:

```
typedef struct bicicleta {
    int id_bicicleta;
    int id_repuesto;
    bool arreglada;
} bicicleta_t;
```

Se pide:

1. Implementar una función que reciba la bandeja de pedidos válidos y cargue la cola_bicicletas_t con una bicicleta por cada pedido, inicializando arreglada en false.
2. Implementar una función que recorra la cola de bicicletas. Para cada una, debe buscar el repuesto correspondiente en la caja (usando su id_repuesto) y, si lo encuentra, extraerlo de la pila y marcar la bicicleta como arreglada.

```
ifndef COLA_BICICLETAS_H#define COLA_BICICLETAS_H

#include <stdbool.h>
```

```
typedef struct bicicleta {
    int id_bicicleta;
    int id_repuesto;
    bool arreglada;
} bicicleta_t;

typedef struct cola_bicicletas cola_bicicletas_t;

/*
 * Pre: -
 * Post: Crea una cola vacía reservada en el heap,
 *        o NULL si falla la reserva.
 */
cola_bicicletas_t* cola_bicicletas_crear();

/*
 * Pre: cola válida.
 * Post: Libera toda la memoria asociada.
 */
void cola_bicicletas_destruir(cola_bicicletas_t* cola);

/*
 * Pre: cola válida.
 * Post: Encola una bicicleta al final.
 */
void cola_bicicletas_encolar(cola_bicicletas_t* cola, bicicleta_t bicicleta);

/*
 * Pre: cola válida y no vacía.
 * Post: Desencola la primera bicicleta y la devuelve.
 */
bicicleta_t* cola_bicicletas_desencolar(cola_bicicletas_t* cola);

/*
 * Pre: cola válida.
 * Post: Devuelve la primera bicicleta sin quitarla, o NULL si está vacía.
 */
bicicleta_t* cola_bicicletas_frente(cola_bicicletas_t* cola);
```

```

/*
 * Pre: cola válida.
 * Post: Devuelve true si la cola está vacía.
 */
bool cola_bicicletas_vacia(cola_bicicletas_t* cola);

#endif

```

Ejercicio de final

Homero está compitiendo en un concurso de comer donas y para ayudarlo a ganar, se quiere ordenar la torre de donas dejando las de sabor frutilla arriba porque no son sus favoritas. Las demás les encantan y le da igual.

Se tiene el siguiente TDA pila_donas que representa la torre de donas:

```

#ifndef PILA_DONAS_H
#define PILA_DONAS_H

#include <stdbool.h>
#include <stddef.h>

#define MAX_SABOR 500

typedef struct dona {
    char* sabor[MAX_SABOR]; // "frutilla", "chocolate", "glaseada"
    bool con_glaseado;
    int tentacion;
} dona_t;

typedef struct pila_donas pila_donas_t;

/*
 * Crea una pila de donas vacía.
 * Devuelve un puntero a la pila creada, o NULL en caso de error.
 */
pila_donas_t* pila_donas_crear();

/*
 * Destruye la pila liberando todos los recursos de la misma.
 */
void pila_donas_destruir(pila_donas_t* pila);

```

```
/*
 * Apila una dona. Devuelve true en éxito, false en error (por ejemplo, memoria
 insuficiente).
 */
bool pila_donas_apilar(pila_donas_t* pila, dona_t* dona);

/*
 * Desapila y devuelve la dona del tope. Devuelve NULL si la pila está vacía.
 */
dona_t* pila_donas_desapilar(pila_donas_t* pila);

/*
 * Indica si la pila está vacía.
 */
bool pila_donas_esta_vacia(const pila_donas_t* pila);

#endif /* PILA_DONAS_H */
```

Se pide

Dada una pila de donas desordenadas, crear una función que procese la pila para que ponga arriba a las donas de frutilla, dejando a las demás en el orden en que fueron encontradas.