Nombre	APELLIDOS (en mayúsculas)	Nota

Listas

3,25 ptos. **Ejercicio 1: Borrar elemento de una lista enlazada.** Dada una lista enlazada y un elemento de tipo `void*`, se pide implementar una función que borre el elemento de la lista, si está presente. La función no devuelve ningún valor, sólo modifica la lista.

El prototipo de la función es:

```
void list_delete_element(List *pl, const void *elem, cmp_fn compare);
```

El tipo de función `cmp_fn` para comparar elementos devuelve `0` solo cuando los elementos comparados son iguales, y se define así:

```
typedef int (*cmp_fn) (const void* el1, const void *el2);
```

Las estructuras de datos a usar son:

```
struct _Node {
  void *info;
  struct _Node *next;
};

typedef struct _Node Node;

struct _List {
  Node *first;
};

typedef struct _List List;
```

Solución del Ejercicio 1.

a) Solución 1: guardar el nodo previo al iterar.

```
void list_delete_element1(List *pl, const void *elem, elem_cmp_fn compare)
    {
  Node *pn, *previous;
  if (!pl || !elem || !compare || list_isEmpty(pl))
    return;

  previous = NULL;
  for (pn = pl->first; pn != NULL; pn = pn->next) {
    if (compare(elem, pn->info) == 0) {
      if (previous) {
        previous->next = pn->next;
      } else {
        pl->first = pn->next;
      }
      free(pn);
      return;
    }
    previous = pn;
  }
}
```

b) Solución 2: dobles punteros.

```
void list_delete_element2(List *pl, const void *elem, elem_cmp_fn compare)
    {
  Node **ppn, *aux;
  if (!pl || !elem || !compare || list_isEmpty(pl))
    return;

  ppn = &(pl->first);

  for (ppn = &(pl->first); (*ppn) != NULL; ppn = &((*ppn)->next)) {
    if (compare(elem, (*ppn)->info) == 0) {
      aux = *ppn;
      *ppn = (*ppn)->next;
      free(aux);
      return;
    }
  }
}
```

c) Solución 3: comparar con siguiente elemento.

```
void list_delete_element3(List *pl, const void *elem, elem_cmp_fn compare)
    {
    Node *pn, *aux;
    if (!pl || !elem || !compare || list_isEmpty(pl))
      return;

    if (compare(pl->first->info, elem) == 0) {
      pn = pl->first;
      pl->first = pn->next;
      free(pn);
      return;
    }

    for (pn = pl->first; pn->next != NULL; pn = pn->next) {
      if (compare(elem, pn->next->info) == 0) {
        aux = pn->next;
```

```
            pn->next = pn->next->next;
            free(aux);
            return;
        }
    }
}
```

Nombre	APELLIDOS (en mayúsculas)	Nota

Árboles

3,25 ptos. **Ejercicio 2.** Escribir el código C de la función `Status tree_prune(BTree *t, int max_depth)`, que recibe un árbol binario `t` (no necesariamente de búsqueda) y lo poda eliminando todos los nodos que se encuentren a una profundidad mayor que `max_depth`.

Se suponen las siguientes estructuras de datos en la definición del TAD `BTree`:

```
typedef struct _BTree {
    BTNode *root;
} BTree;

typedef struct _BTNode {
    void *info;
    struct _BTNode *left;
    struct _BTNode *right;
} BTNode;
```

Y la siguiente función para liberar recursivamente un nodo del árbol y todos sus descendientes:

```
void _tree_recFree(BTNode *pn);
```

Nota. *Se recuerda que la profundidad de un nodo es el número de aristas en el camino que une ese nodo con el nodo raíz del árbol. La profundidad del nodo raíz es* 0.

Solución del Ejercicio 2.

```
Status tree_prune(BTree *t, int max_depth) {
  if (t == NULL) return ERROR;

  if (max_depth < 0) {
    _tree_recFree(t->root);
    t->root = NULL;
  } else {
    _tree_prune_rec(t->root, max_depth);
  }

  return OK;
}

void _tree_prune_rec(BTNode *n, int max_depth) {
  if (n == NULL) return;

  if (max_depth == 0) {
    _tree_recFree(n->left);
    n->left = NULL;
    _tree_recFree(n->right);
    n->right = NULL;
    return;
  }

  _tree_prune_rec(n->left, max_depth-1);
  _tree_prune_rec(n->right, max_depth-1);
}
```

Nombre	APELLIDOS (en mayúsculas)	Nota

Recursión

3,50 ptos. **Ejercicio 3.**

3,25 ptos. a) Proporcionar el código C de una **única** función recursiva que recibe una pila y devuelve una copia de la misma. La pila original no debe modificarse en ningún caso. Se asume que la pila original no es nula, y la interfaz utilizada en clase para el TAD pila.

```
/*
* @brief  This function recursively copies a stack. The original stack is
    not modified
* @param s A pointer to the stack
* @return Return the copy of the stack otherwise return NULL.
*/
Stack * stack_copy_rec (Stack *s);
```

0,25 ptos. b) Identificar cuáles de las llamadas recursivas en la implementación anterior corresponden a recursión de cola, y eliminar la recursión de cola (si hay alguna).

Solución del Ejercicio 3.

a)

```c
#include "stack.h"

Stack *stack_copy_rec (Stack *s) {
    Stack *s_cpy = NULL;
    void *ele;
    Status st;

    //-----base case-------
    if (stack_isEmpty (s) == TRUE) {
        return stack_init ();
    }

    //------General case--------
    ele = stack_pop (s);
    s_cpy = stack_copy_rec (s);

    // Recover the original stack, not necessary CdE
    stack_push (s, ele);

    // Insert the pushed element in the copy stack
    if (s_cpy) {
        st = stack_push (s_cpy, ele);
        if (st == ERROR) {
            stack_free (s_cpy);
            return NULL;
        }
    }

    return s_cpy;
}
```

b) No hay ninguna llamada que sea recursión de cola.