Nombre	APELLIDOS (en mayúsculas)	Nota

Listas

3 ptos. **Ejercicio 1.** Dadas las siguientes definiciones correspondientes a los TAD *List* (lista enlazada simple) y *CList* (lista enlazada circular), ubicadas **en un mismo módulo** (es decir, se tiene acceso interno a ambos TAD):

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

typedef struct _Node Node;
```

```
typedef struct _List List;
typedef struct _CList CList;
```

```
struct _List {
  Node *first;
};
struct _CList {
  Node *last;
};
```

Se pide implementar una función que cree una lista circular a partir de una lista simple, con el prototipo `CList *list_newCList(const List *pl)`, y con los siguientes requisitos:

- La lista simple recibida no debe ser modificada en ningún momento.
- La lista circular creada debe contener los mismos elementos que la lista simple recibida.
- La función debe tener coste lineal $\mathcal{O}(N)$.
- En caso de error, la función debe liberar los recursos reservados y devolver `NULL`.
- Se puede hacer uso de la interfaz estándar tanto de listas como de listas circulares.

Solución del Ejercicio 1.

```
CList *list_newCList(const List *pl) {
  CList *pcl = NULL;
  Node *pn = NULL;

  if (!(pl)) {
    return NULL;
  }

  if (!(pcl = clist_new())) {
    return NULL;
  }

  pn = pl->first;
  while (pn) {
    if (clist_pushBack(pcl, pn->info) == ERROR) {
      clist_free(pcl);
      return NULL;
    }
    pn = pn->next;
  }

  return pcl;
}
```

Nombre	APELLIDOS (en mayúsculas)	Nota

Árboles

3 ptos. **Ejercicio 2.** Tenemos los siguientes tipos:

```
typedef enum { FALSE = 0, TRUE = 1 } Bool;

typedef struct _BSTNode {
  void *info;
  struct _BSTNode *left;
  struct _BSTNode *right;
} BSTNode;

typedef struct _BSTree {
  BSTNode *root;
} BSTree;

typedef Bool (*bool_fn)(void *elem);
```

2 ptos. a) Define una función

```
int tree_count_if(BSTree *tree, bool_fn test);
```

que retorna el número de elementos del árbol para los que el predicado booleano `test` retorna `true`.
Por ejemplo, dadas las funciones:

```
Bool is_positive_number(void *elem) {
  if (!elem)
    return FALSE;
  return *(int *)elem > 0;
}

Bool is_short_string(void *elem) {
  if (!elem)
    return FALSE;
  return strlen(*(char **)elem) < 5;
}
```

la llamada `tree_count_if(tree1, is_positive_number)` nos devolvería el número de números positivos en el árbol de enteros `tree1`; mientras que la llamada `tree_count_if(tree2, is_short_string)` devolvería el número de cadenas de longitud menor que 5 en el árbol de cadenas `tree2`.

1 ptos. b) Un array asociativo, mapa, o diccionario, permite asociar valores a claves, de forma que se pueda acceder de forma eficiente al valor asociado a una clave dada. Por ejemplo, podemos querer asociar los países del mundo (claves) a sus capitales (valores), de forma que, dado un país, podamos obtener rápidamente su capital. Explica verbalmente (no es necesario código), cómo se puede usar un árbol binario de búsqueda para añadir un par clave–valor y para obtener el valor asociado a una clave. Puedes asumir que tanto claves como valores son cadenas de caracteres. ¿Qué complejidad tienen ambas operaciones?

Solución del Ejercicio 2.

a)

```
int tree_count_if_rec(BSTNode *node, bool_fn test);

int tree_count_if(BSTree *tree, bool_fn test) {
  if (!tree || !test)
    return 0;
  return tree_count_if_rec(tree->root, test);
}

int tree_count_if_rec(BSTNode *node, bool_fn test) {
  int res = 0;
  if (!node)
    return 0;
  test(node->info) ? res = 1 : res = 0;
  return res + tree_count_if_rec(node->left) + tree_count_if_rec(node->
      right);
  /* equivalently, but perhaps less clear:
  return test(node->info) + tree_count_if_rec(...) + tree_count_if_rec(...)
  */
}
```

b) Usamos un `struct` con campos `key` y `value`, por ejemplo, `typedef struct {char *key; char *value} keyValuePair`, y creamos un BST con esos structs. Puesto que el campo `info` de un nodo es `void *`, `info` puede contener punteros a estos `structs` (de la misma forma que hemos creado árboles con punteros a Vertex, por ejemplo). La función de comparación del árbol debe comparar sólo el campo clave (luego, implícitamente, no permitimos claves repetidas).
Añadir un `keyValuePair` se hace con `tree_insert`, obtener el valor de una clave se hace con `tree_search`. Es decir, le pasamos un `keyValuePair` con la clave deseada, y `tree_search` nos devolverá el `keyValuePair` con esa clave (si lo encuentra) del árbol. Podemos obtener el valor asociado a la clave trivialmente a partir del par retornado.
La complejidad de ambas operaciones es $O(p)$, donde p es la profundidad del BST, $O(log\ n)$ en el caso de árboles equilibrados.
No es necesario, como puede verse, modificar la estructura de `BSTree` o `BSTNode` en absoluto (y es un grave error hacerlo). Tampoco lo es modificar el uso de los nodos o los hijos de formas "imaginativas" que no se corresponden con su perfectamente definida función como nodos de un árbol binario de búsqueda.

Nombre	APELLIDOS (en mayúsculas)	Nota

Recursión

4 ptos. **Ejercicio 3.**

2,50 ptos. a) Suponga una secuencia de números enteros creciente que, a partir de una determinada posición, se convierte en decreciente. Por ejemplo, la secuencia {0, 4, 6, 8, 7, 5, 1} es una secuencia *creciente-decreciente*. El pivote será el elemento del array en la que cambia el sentido de la secuencia. En el ejemplo anterior el índice del pivote será 3. Proporcione un algoritmo que reciba un array de números enteros y devuelva el índice del pivote. No se considerarán válidas soluciones no eficientes.

Ejemplos:

`Input:` {0, 4, 6, 8, 7, 5, 1}
`Output:` 3
`Input:` {0, 4, 6, 7, 8}
`Output:` 5
`Input:` {10, 8, 5, 2, 8}
`Output:` -

0,50 ptos. b) ¿Cúal es el coste del algoritmo anterior? Justifique su respuesta.

1 ptos. c) El siguiente algoritmo recursivo (pseudocódigo) realiza una búsqueda en profundidad en un grafo G partiendo de un nodo v:

```
DFS(G, v)
    set_vertex_label(v, BLACK)
    for all w that are in G.adjacent_vertex(v) do:
        if vertex w is not labeled as BLACK then:
            DFS(G, w)
```

Suponed que el algoritmo anterior se aplica a un grafo regular G en el que **todo nodo** v del grafo tienen 4 vértices adyacentes: `UP(v)`, `DOWN(v)`, `LEFT(v)`, `RIGHT(v)`, que siempre se exploran en ese orden.

- Identificar la recursión de cola, si la hubiese. Justificar la respuesta.
- Eliminar la recursión de cola, si la hubiese.

Solución del Ejercicio 3.

a) Este problema se resolvió en clase y, además, la solución está incluída en la hoja de problemas de Moodle.

```
int _creciente_decreciente(int a[], int first, int last, int size) {
    int m;
    if (first > last) return -1;

    m = (first + last) / 2;          // índice medio
    if (m == size - 1)               // monotona creciente
        return size;
    else if (m == 0)                 // monotona decreciente
        return -1;

    else if (a[m] > a[m+1] && a[m] > a[m-1])
        return m;
    else if (a[m] < a[m+1])
        return _creciente_decreciente(a, m+1, last, size);
    else
        return _creciente_decreciente(a, first, m-1, size);
}
```

b) El bucle `for` implica 4 llamadas recursivas. Solo la última es recursión de cola

```
DFS(G, v)
    set_vertex_label(v, BLACK)
    // Bucle for de forma explícita
    w = UP(v)
    if vertex w is not labeled as BLACK then:
        DFS(G, w)
    w = DOWN(v)
    if vertex w is not labeled as BLACK then:
        DFS(G, w)
    w = LEFT(v)
    if vertex w is not labeled as BLACK then:
        DFS(G, w)
    w = RIGHT(v)
    if vertex w is not labeled as BLACK then:
        DFS(G, w)    // Recursión de cola
```

Eliminación de la recursión de cola. Codigo no estructurado con sentencia `goto`:

```
DFS(G, v)
     LABEL:
        set_vertex_label(v, BLACK)

        w = UP(v)
        if vertex w is not labeled as BLACK then:
            DFS(G, w)
        w = DOWN(v)
        if vertex w is not labeled as BLACK then:
            DFS(G, w)
        w = LEFT(v)
        if vertex w is not labeled as BLACK then:
            DFS(G, w)

        v = RIGHT(v)
        if vertex v is not labeled as BLACK then:
            goto LABEL
```

Eliminación de la recursión de cola. Código estructurado:

```
DFS(G, v)
    while v is not labeled as BLACK:
        set_vertex_label(v, BLACK)

        w = UP(v)
        if vertex w is not labeled as BLACK then:
            DFS(G, w)
        w = DOWN(v)
        if vertex w is not labeled as BLACK then:
            DFS(G, w)
        w = LEFT(v)
        if vertex w is not labeled as BLACK then:
            DFS(G, w)

        v = RIGHT(v)
```