Nombre	APELLIDOS (en mayúsculas)	Nota

TAD

1,50 ptos. **Ejercicio 1.** Se desea implementar un TAD que represente una clase de estudiantes (TAD *Class*), que estará compuesta por un máximo de 200 estudiantes (TAD *Student*). Cada estudiante está definido por su nombre (máximo 1024 caracteres), el año en que comenzó la carrera, y su expediente académico (TAD *Register*). El expediente viene dado por la especialidad (máximo 1024 caracteres), y las como máximo 50 asignaturas cursadas (TAD *Subject*). Cada asignatura contiene el nombre de la materia y el del profesor (1024 caracteres cada uno), el número de créditos, y la calificación obtenida.

0,50 ptos. a) Escribir en C las estructuras de datos y las definiciones de nuevos tipos necesarias para implementar los TAD anteriores, garantizando al máximo la abstracción de los detalles de la implementación. Indicar en qué fichero se localizaría cada una de ellas.

1 ptos. b) Escribir el código C de una función que, dada una clase y el nombre de un estudiante, devuelva el número de créditos aprobado por ese estudiante (asumiendo que la calificación debe ser mayor o igual a 5 para aprobar cada asignatura), o `-1` si no se encuentra el estudiante. Indicar en qué fichero se localizaría la nueva función.

Nota. *Se puede asumir la existencia de primitivas básicas en todos los TAD (acceso directo a los campos mediante* setters *y* getters*). Si se considera necesario, se pueden implementar primitivas auxiliares en cualquier TAD, indicando dónde se localizarían.*

Solución del Ejercicio 1.

a) **TAD *Subject***

En `subject.c`:

```
struct _Subject {
  char name[MAX_WORD];
  char teacher[MAX_WORD];
  int credits;
  float mark;
};
```

En `subject.h`:

```
typedef struct _Subject Subject;
```

TAD *Record*

En `record.c`:

```
struct _Record {
  char speciality[MAX_WORD];
  Subject *subjects[MAX_SUBJECTS];
  int n_subjects;
};
```

En `record.h`:

```
typedef struct _Record Record;
```

TAD *Student*

En `student.c`:

```
struct _Student {
  char name[MAX_WORD];
  int starting_year;
  Record *record;
};
```

En `student.h`:

```
typedef struct _Student Student;
```

TAD *Class*

En `class.c`:

```
struct _Class {
  Student *students[MAX_STUDENTS];
  int n_students;
};
```

En `class.h`:

```
typedef struct _Class Class;
```

b) En `record.c`:

```
int record_credits(Record *r) {
  int i, credits = 0;

  if (!(r)) {
    return 0;
  }

  for (i = 0; i < r->n_subjects; i++) {
```

```
    if (subject_get_mark(r->subjects[i]) >= 5.0) {
      credits += subject_get_credits(r->subjects[i]);
    }
  }

  return credits;
}
```

En `class.c`:

```
int class_student_credits(Class *c, char *name) {
  int i = 0;

  for (i = 0; i < c->n_students; i++) {
    if (!(strcmp(name, student_get_name(c->students[i])))) {
      return record_credits(student_get_record(c->students[i]));
    }
  }
  return CREDITS_INVALID;
}
```

Nombre	APELLIDOS (en mayúsculas)	Nota

Pilas y expresiones

2 ptos. **Ejercicio 2.**

1,50 ptos. a) Proporcione el código C con control de errores de una función `Status decimal2base (int number, int base, char *str)` que, mediante el algoritmo de división repetida por una base y acumulación de restos y haciendo uso de una pila, proporcione la representación como una cadena de caracteres de un número entero positivo en una base menor de 10. Asuma que `str` tiene capacidad suficiente para almacenar la representación del número entero en la base correspondiente. Utilice la interfaz del TAD `Stack` vista en clase y asuma que la pila tiene capacidad suficiente para almacenar todos los dígitos del número entero. No se considerarán válidas soluciones no eficientes ni correctamente estructuradas y/o modularizadas o con código repetitivo o de difícil lectura.

0,50 ptos. b) Transformar la siguiente expresión prefija a forma infija, mostrando cada paso de la conversión. Solo está permitido el uso de una única pila auxiliar: `* / + A B + C D E`.

Símbolo	Pila

Solución del Ejercicio 2.

a)

```
Status recover_errror (Stack *s) {
  while (!stack_isEmpty(s)) {
      free(stack_pop(s));
  }
  stack_free (s);
  return ERROR;
}

Status decimal2base(int number, int base, char *str) {
  Stack *s = NULL;
  int *remainder = NULL;
  int i = 0;
  int *tmp = NULL;
  Status st = OK;

  if (!str) return ERROR;
  s = stack_init();                          // init auxiliary stack
  if (!s)
    return ERROR;

  while (number >= base && st == OK) {
    remainder = (int *) malloc(sizeof(int)); // memory for element
    if (remainder) {
      *remainder = number % base;
      number = number / base;
      stack_push(s, remainder);              // No error buffer overflow
    } else
      st = ERROR;                            // memory allocation error
  }
  if (st == ERROR)
    return recover_error(s);
  str[i++] = number + '0';                   // most significative digit
                                             //  convert integer to char
  while (!stack_isEmpty(s)) {
      tmp = (int *) stack_pop(s);
      str[i++] = *tmp + '0';                 // convert integer to char
      free(tmp);                             // frre stack element
  }
  str[i] = '\0';                            // Insert EOS in output char
  stack_free(s);
  return OK;
}
```

b)

Símbolo	Pila
	-
*	*
/	*, /
+	*, /, +
A	*, + , A
B	* , (A + B)
+	* , (B + A), +
C	* , (B + A), +, C
D	((B + A) / (D / C))
E	(((B + A) / (D / C)) * E)
;	-

Cuando se lee el símbolo de fin de cadena `';'` se realiza un `pop`, si la pila queda vacía la exresión infija es correcta. La expresión final es por tanto `(((A + B)* (C + D))* E)`.

Nombre	APELLIDOS (en mayúsculas)	Nota

Colas

1,50 ptos. **Ejercicio 3.**

1,20 ptos. a) Define una función `void *queue_max(Queue *pq, int (*elem_cmp)(const void *, const void *))` que retorna el elemento máximo de la cola `pq` usando la función de comparación `elem_cmp` que se le pasa como argumento (y que deberá seguir la convención de C para funciones de comparación). La función `queue_max` **debe** acceder a la implementación de la cola, sin modificar en ningún momento su estado.

0,30 ptos. b) Ilustra con un ejemplo cuál sería una posible función de comparación que se pueda pasar como argumento a `queue_max`.

Solución del Ejercicio 3.

a)

```
void *queue_max(Queue *pq, int (*elem_cmp)(const void *, const void *)) {
  int i;
  void *max = NULL;
  if (!pq || !elem_cmp || queue_isEmpty(pq))
    return NULL;
  max = pq->data[pq->front];
  for (i = (pq->front + 1) % MAX_QUEUE; i != pq->rear;
       i = (i + 1) % MAX_QUEUE) {
    if (elem_cmp(max, pq->data[i]) < 0)
      max = pq->data[i];
  }
  return max;
}
```

b)

```
// Sample comparison function, for a queue of integers
int int_cmp(const void *c1, const void *c2) {
  if (!c1 || !c2)
    return 0;

  return (*(int *)c1 - *(int *)c2);
}
```

Nombre	APELLIDOS (en mayúsculas)	Nota

Listas

1,50 ptos. **Ejercicio 4.** Asumiendo la estructura y la interfaz estándar para el TAD *CList* (lista enlazada circular simple), se pide implementar la siguiente función para reemplazar elementos:

```
Status clist_replace(CList *pcl, const void *e, int (*check_elem)(const void *))
```

Esta función sustituirá la primera aparición de un cierto elemento (identificado porque la función recibida `check_elem` devuelve `0`) por el elemento `e` recibido. La función deberá devolver `OK` si todo fue correctamente, y `ERROR` en caso de error o si no se encuentra el elemento que se quiere reemplazar.

Solución del Ejercicio 4.

```
Status clist_replace(CList *pcl, const void *e, int (*check_elem)(const void *))
     {
  Node *pn = NULL;

  if ((!(pcl)) || (!(e)) || (!(check_elem)) || (clist_isEmpty(pcl) == TRUE)) {
    return ERROR;
  }

  pn = pcl->last->next;
  do {
    if (!(check_elem(pn->info))) {
      pn->info = (void *)e;
      return OK;
    }

    pn = pn->next;
  } while (pn != pcl->last->next);

  return ERROR;
}
```

Nombre	APELLIDOS (en mayúsculas)	Nota

Árboles

1,50 ptos. **Ejercicio 5.** Queremos definir un tipo de datos `AMap` (para AssociativeMap), que nos permite asociar claves a valores asociados a esas claves. 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. Puedes asumir que tanto claves como valores son cadenas de caracteres.

Para implementarlo, definimos un `AMap` mediante un árbol binario de búsqueda, que almacene pares claves–valor, teniendo en cuenta solo las claves para ordenar los elementos (por tanto, los valores asociados a cada clave **no** se usan para comparar dos pares):

```
// in elements.h
//////////////////
// function type to print a void* elem to a FILE* stream
typedef int (*elem_print_fn)(FILE *, const void *elem);
// function type to compare two elements of type void
//  it assumed this function uses the C convention for the return value of a
//  comparison function
typedef int (*elem_compare_fn)(const void *elem1, const void *elem2);

// in bstree.h
//////////////
#include "elements.h"
typedef struct _BSTree BSTree;
// function that initializes a binary search tree with a print function for tree
// elements and a comparison function used for the ordering of the elements in
// the tree
BSTree *tree_init(elem_printf_fn print_elem, elem_compare_fn compare_elem);

// function to insert an element elem in a tree
Status tree_insert(BSTree *tree, const void *elem);

// function that searchs for an element elem in the tree, and returns the
// element in the tree (a node's info field), if any, for which the
// tree's comparison function returns 0
void *tree_find(BSTree *tree, const void *elem);

// function to free a tree
void tree_destroy(BSTree *tree);

// in amap.h
//////////////
typedef struct _AMap AMap;
// prototipos de función necesarios para el examen....

// in amap.c
//////////////
#include "amap.h"
#include "bstree.h"
#include "elements.h"

struct _AMap {
  BSTree *map;
}
```

a) Define un tipo apropiado `KeyValuePair` para almacenar los pares clave–valor.
b) Define la función de comparación entre dos `KeyValuePair`'s, `int cmp_key(void *el1, void *el2)`, necesaria para poder gestionar los pares clave–valor en el árbol utilizado por el `AMap`.
c) Define la función `Status amap_add(AMap *map, char *key, char *value)` para añadir una clave al árbol del `AMap` con su valor asociado.
d) Define la función `char *amap_get(AMap *map, char *key)`, que devuelve el valor asociado a la clave `key` o `NULL` si la clave no se encuentra en el árbol.

Puedes asumir funciones auxiliares adicionales para inicializar, imprimir y liberar `KeyValuePairs`, y para inicializar y liberar el árbol del `AMap`.

¿Qué complejidad tienen ambas operaciones?

Solución del Ejercicio 5. Nota: Para la solución, solo se requieren las tres funciones solicitadas.

a)

```
typedef struct {
  char *key;
  char *value;
} KeyValuePair;

KeyValuePair *key_value_init(const char *key, const char *value) {
  KeyValuePair *pkv = NULL;

  if (!key || !value)
    return NULL;
  pkv = malloc(sizeof(KeyValuePair));
  if (!pkv)
    return NULL;
  pkv->key = key;
  pkv->value = value;
  return pkv;
}

void key_value_free(KeyValuePair *pkv) {
  // calling function should set it to NULL
  free(pkv);
}

int key_value_print(FILE *pf, void *vpkv) {
  KeyValuePair *pkv;
  if (!vpkv)
    return -1;
  pkv = (KeyValuePair *)vpkv;
  if (!(pkv->key) || !(pkv->value))
    return -1;
  return printf("%s: %s, ", pkv->key, pkv->value);
}

int key_compare(const void *kv1, const void *kv2) {
  if (!kv1 || !kv2)
    return 0;
  // note strcmp  has undefined behavior if either key is NULL;
  // but no KeyValuePair is created with a NULL key
  return strcmp(((KeyValuePair *)kv1)->key, ((KeyValuePair *)kv2)->key);
}

AMap *amap_init() {
  AMap *amap;
  amap = malloc(sizeof(AMap));
  if (!amap)
    return NULL;
  amap->map = tree_init(key_value_print, key_compare);
  if (!(amap->map)) {
    free(amap);
    return NULL;
  }
  return amap;
}

Status amap_add(AMap *amap, char *key, char *value) {
  KeyValuePair *pkv;
  Status st = OK;
```

```
  if (!amap || !key || !value)
    return ERROR;
  pkv = key_value_pair_init(key, value);
  if (!pkv)
    return ERROR;
  st = tree_insert(amap->map, pkv);
  if (st == ERROR) {
    key_value_free(pkv);
  }
  return st;
}

char *amap_get(AMap *amap, char *key) {
  KeyValuePair pkv_dummy, *pkv = NULL;
  char *value = NULL;

  if (!amap || !key)
    return NULL;

  // in order to use the tree's search function, we need to search with an
  // element of the same type as nodes' info fields (do we?)
  pkv_dummy.key = key;
  pkv_dummy.value = "";

  pkv = tree_find(amap->tree, &pkv_dummy);

  if (pkv)
    value = pkv->value;

  return value; // possibly NULL
}
```

b) La complejidad es $\mathcal{O}(p)$, donde p es la profundidad del BST; $\mathcal{O}(\log n)$ si se trata de un BST equilibrado de n elementos.

Nombre	APELLIDOS (en mayúsculas)	Nota

Recursión

2 ptos. **Ejercicio 6.**

1,50 ptos. a) La función `stack_merge()` devuelve una pila ordenada en orden ascendente, considerado desde la base de la pila, a partir de dos pilas ordenadas con el mismo orden. `stack_merge()` recibe tres parámetros de entrada: las dos pilas ordenadas y una función que dados dos elementos cualesquiera determine cuál es mayor.
Proporcione el código C, con control de errores, del procedimiento **recursivo** `stack_merge()`. La función vaciará las dos pilas originales. Considere la interfaz habitual del TAD `Stack` pero **asumiendo** que la pila devuelta tiene capacidad suficiente para almacenar todos los elementos de las pilas. No se considerarán válidas soluciones no eficientes. No se permite acceder a la estructura de datos, deben usarse solo las funciones de la interfaz pública.

0,50 ptos. b) A diferencia de los árboles binarios, en los árboles ternarios un nodo puede tener, a lo sumo, tres hijos. Considere el siguiente algoritmo:

```c
#define MAX_CHILD 3
typedef struct _Node {
    void *info;
    struct _Node *child[MAX_CHILD];
} Node;

typedef struct {
    Node *root;
} TernaryTree;

typedef Boolean (*t_boolean)(void *);

void _fun_unknown(Node *node, t_boolean f, int *total){
    int i;
    if f(node->data)
        (*total)++;
    for (i=0; i<MAX_CHILD; i++) {
        if (node->child[i])
            _fun_unknown(node->child[i], f, total);
    }
}

int fun_unknown(TernaryTree *t, t_boolean f) {
    int total = 0;
    if (!t || !f) return -1;
    if (t->root) {
        _fun_unknown(t->root, f, &total);
        return total;
    } else return 0;
}
```

- Explique brevemente qué hace y en qué orden procesa los nodos el algoritmo. Si es necesario, ayúdese con un ejemplo.
- Identifique la recursión de cola, si la hubiese. Justifique la respuesta.
- Elimine la recursión de cola, si la hubiese.

Solución del Ejercicio 6.

a)

- **Solución 1.** Utiliza una función privada. La función envolvente `stack_merge()` realiza el control de los parámetros de entrada de la función y crea la pila de retorno. Si la pila se ha creado se invoca a la función privada `_stack_merge()`.

```
3  typedef int (*t_greater)(void *, void*);
4
5  void _stack_merge(Stack *s1, Stack *s2, Stack *s3, t_greater f) {
6      void *ele;
7
8      /*---- Base case: Both stacks are empty-----*/
9      if (stack_is_empty(s1) && stack_is_empty(s2))
10         return;
11     /*---- General case: At least stack is not empty-----*/
12     if (!stack_is_empty(s1) && !stack_is_empty(s2)){
13         if (f(stack_top(s1), stack_top(s2)) > 0) {
14             ele = stack_pop(s1);
15         else
16             ele = stack_pop(s2);
17     }
18     else if stack_is_empty(s1) {
19         ele = stack_pop(s2);
20     }
21     else
22         ele = stack_pop(s1);
23
24     _stack_merge(s1, s2, s3, f);
25     stack_push(s3, ele); // importante DESPUÉS de la recursión
26     return;
27 }
28
29 Stack *stack_merge(Stack *s1, Stack *s2, t_greater f){
30     Stack *s3 = NULL;
31
32     if (!s1 || !s2 || !s3 || !f)
33         return NULL;
34
35     s3 = stack_ini();
36     if (!s3)
37         return NULL;
38
39     _stack_merge(s1, s2, s3, f);
40     return s3;
41 }
```

Es importante que cuando se inserte el elemento en la pila de salida (sentencia 25 del código anterior) se haga **después** de la llamada recursiva para que la pila de salida quede ordenada de menor a mayor.

- **Solución 2.** Sin función privada. La pila de retorno se crea en el caso base de la recursión. Presentamos primero una implementación simplificada, sin control de errores y, a continuación con control de errores
  - Sin control de errores (solución incompleta)

```
46 typedef int (*t_greater)(void *, void*);
47
48 Stack *stack_merge(Stack *s1, Stack *s2, t_greater f) {
49     void *ele;
50     Stack *s3;
51
52     /*-- Base case: Both stacks are empty -----*/
```

```
53      if (stack_is_empty(s1) && stack_is_empty(s2)) {
54          s3 = stack_init()
55          return s3;
56      }
57      /*-- General case: At least one stack is not empty --*/
58      if (!stack_is_empty(s1) && !stack_is_empty(s2)){
59          if (f(stack_top(s1), stack_top(s2)) > 0) {
60              ele = stack_pop(s1);
61          }
62          else {
63              ele = stack_pop(s2);
64          }
65      }
66      else if stack_is_empty(s1) {
67          ele = stack_pop(s2);
68      }
69      else {
70          ele = stack_pop(s1);
71      }
72
73      s3 = stack_merge(s1, s2, f);
74      stack_push(s3, ele);
75      return s3;
76  }
```

- Con control de errores. En la solución anterior **no se puede garantizar** que se inicialice la pila en `stack_init()`. Por tanto, hay que incluir el oportuno control de errores. Si no se consigue inicializar la pila habría que devolver los elementos extraídos a su pila de origen. Para poder restituir el elemento extraído a su pila original en el caso de `s3` sea `NULL` necesitamos conocer de que pila se extrajo el elemento, `s1` o `s2`. Para ello se utiliza la variable `s_extract`.

```
82  typedef int (*t_greater)(void *, void*);
83
84  Stack *stack_merge(Stack *s1, Stack *s2, t_greater f) {
85      void *ele;
86      Stack *s3;
87      Stack *s_extract = NULL;
88
89      /*-- Base case: Both stacks are empty -----*/
90      if (stack_is_empty(s1) && stack_is_empty(s2)) {
91          s3 = stack_init()
92          return s3;
93      }
94      /*-- General case: At least one stack is not empty --*/
95      if (!stack_is_empty(s1) && !stack_is_empty(s2)){
96          if (f(stack_top(s1), stack_top(s2)) > 0) {
97              ele = stack_pop(s1);
98              s_extract = s1;     // pila de la que se extrae
99          }
100         else {
101             ele = stack_pop(s2);
102             s_extract = s2;
103         }
104     }
105     else if stack_is_empty(s1) {
106         ele = stack_pop(s2);
107         s_extract = s2;
108     }
109     else {
110         ele = stack_pop(s1);
```

```
111         s_extract = s1;
112     }
113     s3 = stack_merge(s1, s2, f);
114     if (s3)                     // s se inicializó correctamente
115         stack_push(s3, ele);
116     else                        // sino devuelvo el elemento a su pila
117         stack_push(s_extract, ele);
118     return s3;
119 }
```

b)

- El procedimiento recursivo `fun_unknown (TernaryTree *t, t_boolean f)` cuenta el número de nodos del árbol `t` que satisfacen la condición lógica estipulada por la función booleana `f`. Para ello el árbol se explora en profundidad: para todo nodo distinto de `NULL` primero se explora su subárbol indexado en el array por 0, luego el 1 y finalmente el 2.
- El bucle `for` implica tres llamadas recursivas. Solo la última es recursión de cola.

```
int _fun_unknown(Node *node, t_boolean f, int *total){
    int i;
    if f(node->data)
        (*total)++;
    if (node->child[0])
            _fun_unknown(node->child[0], f, total);
    if (node->child[1])
            _fun_unknown(node->child[1], f, total);
    if (node->child[2])
            _fun_unknown(node->child[2], f, total);   // Recursion de
               cola
}
```

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

```
void _fun_unknown(Node *node, t_boolean f, int *total){
    int i;
 LABEL:
    if f(node->data)
        (*total)++;
    if (node->child[0])
            _fun_unknown(node->child[0], f, total);
    if (node->child[1])
            _fun_unknown(node->child[1], f, total);
    if (node->child[2]) {
        node = node->child[2];
        goto LABEL;
     }
}
```

Código estructurado:

```
void _fun_unknown(Node *node, t_boolean f, int *total){
    int i;
    while (node) {
        if f(node->data)
            (*total)++;
        if (node->child[0])
            _fun_unknown(node->child[0], f, total);
        if (node->child[1])
            _fun_unknown(node->child[1], f, total);
        node = node->child[2];
    }
}
```