Nombre	APELLIDOS (en mayúsculas)	Nota

TAD

3 ptos. **Ejercicio 1.** Se desea implementar un TAD que represente un partido de tenis (TAD *Match*), que estará compuesto por el nombre del primer y del segundo jugador (máximo 1024 caracteres cada uno), el año del partido, y el conjunto de como máximo 5 sets disputados (TAD *Set*). Cada set está definida por la lista de como máximo 13 juegos que han tenido lugar (TAD *Game*). Por último, cada juego contiene los puntos ganados por el primer jugador, los puntos ganados por el segundo jugador, y la duración en minutos del juego.

1 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.

2 ptos. b) Escribir el código C de una función que, dado un partido, devuelva un entero indicando el jugador ganador (`1` o `2`), o `0` en caso de error. Para ello, se considera que gana el partido el que más sets ha ganado, gana el set el que más juegos ha ganado dentro del set, y gana el juego el que más puntos ha ganado dentro del juego. En caso de empate a cualquier nivel, se dará por vencedor al primer jugador (en realidad, en un partido real no se puede empatar a ningún nivel). 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 *Game***

En game.c:

```
struct _Game {
  int points1;
  int points2;
  int duration;
};
```

En game.h:

```
typedef struct _Game Game;
```

TAD *Set*

En set.c:

```
struct _Set {
  Game *games[MAX_GAME];
  int n_games;
};
```

En set.h:

```
typedef struct _Set Set;
```

TAD *Match*

En match.c:

```
struct _Match {
  char player1[MAX_WORD];
  char player2[MAX_WORD];
  int year;
  Set *sets[MAX_SET];
  int n_sets;
};
```

En match.h:

```
typedef struct _Match Match;
```

b) En types.h (o similar):

```
typedef enum { INV_PLAYER = 0, PLAYER_1, PLAYER_2 } Player;
```

En match.c:

```
Player match_winner(Match *m) {
  int i, sets_1 = 0, sets_2 = 0, winner;

  if (!(m)) {
    return INV_PLAYER;
  }

  for (i = 0; i < m->n_sets; i++) {
    winner = set_winner(m->sets[i]);
    if (winner == PLAYER_1) {
      sets_1++;
    } else if (winner == PLAYER_2) {
      sets_2++;
    }
  }
```

```
  if (sets_1 >= sets_2) {
    return PLAYER_1;
  } else {
    return PLAYER_2;
  }
}
```

En set.c:

```
Player set_winner(Set *s) {
  int i, games_1 = 0, games_2 = 0, winner;

  if (!(s)) {
    return INV_PLAYER;
  }

  for (i = 0; i < s->n_games; i++) {
    winner = game_winner(s->games[i]);
    if (winner == PLAYER_1) {
      games_1++;
    } else if (winner == PLAYER_2) {
      games_2++;
    }
  }

  if (games_1 >= games_2) {
    return PLAYER_1;
  } else {
    return PLAYER_2;
  }
}
```

En game.c:

```
Player game_winner(Game *g) {
  if (!(g)) {
    return INV_PLAYER;
  }

  if (g->points1 >= g->points2) {
    return PLAYER_1;
  } else {
    return PLAYER_2;
  }
}
```

Nombre	APELLIDOS (en mayúsculas)	Nota

Pilas y expresiones

4 ptos. **Ejercicio 2.**

3 ptos. a) La función `stack_merge()` crea una pila ordenada en orden descendente (de mayor a menor) a partir de dos pilas ordenadas en orden ascendente (de menor a mayor). Suponed, además, que los elementos de las pilas son distinguibles. Es decir, que si los elementos de una de las dos pila satisfacen una determinada condición booleana, los de la otra pila no.
Por tanto, `stack_merge()` recibe cuatro parámetros de entrada: las dos pilas ordenadas, una función que dados dos elementos cualesquiera determine cual es mayor y una segunda función booleana que se evalúa como `True` para los elementos de la primera pila y `False` para los de la segunda.
Proporcione el código, con control de errores, de la función `stack_merge()`. En caso de éxito la función vaciará las dos pilas originales. Sin embargo, si se produjese un error, `stack_merge()` devolverá `NULL` y dejará las dos pilas en su estado original. Asuma la interfaz habitual del TAD *Stack* con sus condiciones de error. *No se permite acceder a la estructura de datos, deben usarse solo las funciones de la interfaz pública.*
Ejemplo:
Suponga, por ejemplo, que tiene una primera pila ordenada con enteros pares y una segunda con impares y se dispone de las funciones:

```
Boolean es_par(void *item) {
    if (*(int*)item % 2 == 0)
        return True;
    return False;
}
int es_mayor(void *e1, void *e2){
    return ( *(int*)e1 - *(int*)e2);
}
```

Entonces cuando se invocase a `stack_merge(s1, s2, es_mayor, es_par)`, en caso de éxito, se obtendría una pila ordenada de mayor a menor con los elementos de ambas.
Atención:
No se considerarán válidas soluciones no eficientes ni correctamente estructuradas y/o modularizadas o con código repetitivo o de difícil lectura.

1 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;`.

Símbolo	Pila

Solución del Ejercicio 2.

a)
- Este problema está resuelto (pseudocódigo) en el curso de Moodle y, además, se implementó en la segunda práctica P2.
- La única diferencia entre esta solución y el pseudocódigo de los apuntes, es que ahora hay que tener en cuanta la posibilidad de que se produzca un *overflow* en la pila `s3`: no podamos insertar un elemento en `s3` porqué la pila estuviese llena. Debemos capturar el retorno de `stack_push(s3, e)` y, en caso de `ERROR`, invocar a la función `restore_merged_stacks()` para recuperar las pilas originales `s1` y `s2`.
- Date cuenta que en la función `restore_merged_stacks()` primero devolvemos el elemento que no conseguimos insertar en `s3` a la pila que le corresponda y, después, vaciar la pila `s3`.

```
typedef int (*t_cmp)(void *, void *);
typedef Boolean (*t_bool)(void *);

void *restore_merged_stacks(Stack *, Stack *, Stack *, void *, t_bool);

Stack *stack_merge(Stack *s1, Stack *s2, t_cmp f_cmp, t_bool f_bool) {
  Stack *s3, *ps = NULL;
  void *e = NULL;
  Status st = OK;

  if (!s1 || !s2 || !f_cmp || !f_bool)  return NULL;

  s3 = stack_ini();                              // init output stack
  if (s3 == NULL)  return NULL;

  // Merge the stacks until one is emptied
  while (!stack_is_empty(s1) && !stack_is_empty(s2) && st == OK) {
    if (f_cmp(stack_top(s1), stack_top(s2)) > 0) {
      e = stack_pop(s1);
      st = stack_push(s3, e);                   // Check buffer overflow error
    } else {
      e = stack_pop(s2);
      st = stack_push(s3, e);                   // Check buffer overflow error
    }
  }

  if (st == ERROR)                              // Error control
    return restore_merged_stacks(s1, s2, s3, e, f_bool);

  if (stack_is_empty(s1)) {                    // Detect the non-emptied stack
    ps = s2;
  } else  ps = s1;

  while (!stack_is_empty(ps) && st == OK) { // Pop from non-emptied stack
    e = stack_pop(ps);
    st = stack_push(s3, e);                     // Check buffer overflow error
  }

  if (st == ERROR)                              // Error control
    return restore_merged_stacks(s1, s2, s3, e, f_bool);
  else
    return s3;
}

void *restore_merged_stacks(Stack *s1, Stack *s2, Stack *s3,
                            void *e, t_bool f_bool) {
```

```
  if (f_bool(e))                              // push poped element
    stack_push(s1, e);
  else stack_push(s2, e);

  while (!stack_is_empty(s3)) {               // Empty the merged-stack
    e = stack_pop(s3);
    if (f_bool(e))
        stack_push(s1, e);
    else stack_push(s2, e);
  }
  stack_free(s3);
  return NULL;
}
```

b) Existen diferente algoritmos:

- Aplicar el mismo algoritmo de traducir de posfijo a infijo pero procesando la expresión de derecha a izquierda
- Procesar la expresión de izquierda a derecha con el algoritmo cuyo pseudocódigo se muestra a continuación:

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

Explicación:

- En el diagrama anterior, los elementos de la pila se han separado por comas.
- Después de leer el caracter `'D'` la pila solo contiene **un elemento**.
- Cuando se lee el símbolo de fin de cadena `';'` se realiza un `pop`, si la pila queda vacía la expresión prefija era correcta. La expresión final infija es `((B + A)* (C / D))`.

Pseudocódigo:

- **No era necesario** incluirlo en la respuesta. Pero tú solución debe respetarlo. Nota el `while` anidado: debido a él después de insertar el caracter `'D'` la pila sólo tiene un elemento.
- No sería correcto un algoritmo que no incluyese el bucle `while` antes insertar un operando en la pila. Puedes comprobarlo con la expresión prefijo `*/+A B + C D E` cuya traducción infijo es `(((A + B)/ (C + D))* E)`.

```
prefix_infix (String str):

 s = stack_ini()

 while str[i] != EOS:
   if es_operador(str[i]):
     stack_push(s, str[i])

   else if es_operando(str[i]):
     operando2 = str[i]
     while !stack_is_empty(s) && es_operando(stack_top(s)):
        operando1 = stack_pop()
        operador = stack_pop()
        operando2 = do_infix_expr(operando1, operador, operando2)

     stack_push(s, operando2)
```

```
    return stack_pop()                    // Fin de cadena
```

Nombre	APELLIDOS (en mayúsculas)	Nota

Colas

3 ptos. **Ejercicio 3.** Queremos modificar la implementación de cola vista en clase:

```
#define MAX_QUEUE 8
struct _Queue {
  void *data[MAX_QUEUE];
  int front;
  int rear;
};
typedef struct _Queue Queue;
```

para permitir que el tamaño de la cola crezca cuando se llena. Para ello, usaremos la siguiente definición alternativa:

```
#define QUEUE_INIT_SIZE 8
#define QUEUE_GROWTH_FACTOR 2

struct _Queue {
  void **data;
  int front;
  int rear;
  int capacity;
};
typedef struct _Queue Queue;
```

Al inicializar la cola, `q->data` se reserva dinámicamente con tamaño `QUEUE_INIT_SIZE`, valor que también se asigna a `q->capacity`. Cuando se agota la capacidad reservada, se usará una función `queue_grow` que: modifica la cola para que `q->data` tenga tamaño `q->capacity * QUEUE_GROWTH_FACTOR`; actualiza `q->capacity` con el nuevo tamaño; y deja la cola modificada en un estado equivalente al de antes de su llamada: con los mismos elementos, en el mismo orden.

Se pide:

2 ptos. a) Implementa la función `Status queue_grow(Queue *q)` que actualiza la cola de la manera descrita.

1 ptos. b) Modifica el código de la función `Status queue_push(Queue *q, const void * elem)` para que use `queue_grow` cuando sea necesario.

```
Status queue_push(Queue *q, const void *elem) {
  Status st = OK;

  if (!q || !elem)
    return ERROR;

  // Añade TU CÓDIGO AQUÍ para usar queue_grow
  // cuando no queda espacio disponible

  q->data[q->rear] = (void *)elem;
  q->rear = (q->rear + 1) % q->capacity;
  return OK;
}
```

Solución del Ejercicio 3.

a)

```
Status queue_grow(Queue *q) {
  void **new_data = NULL;
  int i, j, old_size, new_size;

  if (!q)
    return ERROR;

  old_size = q->capacity;
  new_size = q->capacity * QUEUE_GROWTH_FACTOR;

  // allocate new memory
  // note: realloc wouldn't work, as it may break the circular structure
  // of the queue
  new_data = malloc(sizeof(void *) * new_size);

  if (!new_data)
    return ERROR;

  // copy q->data to new_data
  // note: see alternative solution with realloc and memmove
  j = 0;
  for (i = q->front; i != q->rear; i = (i + 1) % old_size) {
    new_data[j++] = q->data[i];
  }

  free(q->data);
  q->data = new_data;
  q->front = 0;
  q->rear = j;
  q->capacity = new_size;
  return OK;
}
```

Errores más comunes:

- El error más común con diferencia ha sido no darse cuenta de que `realloc`, sin más, no funciona, porque deja la cola en un estado inválido. El estado de la cola será inválido siempre que `q->front != 0` en el momento en el que se llena `q->data` (es decir, siempre que para recorrer la cola, tengamos que llegar al final de `q->data` y continuar desde el principio del array). Por tanto, es necesario copiar los datos para asegurarse de que la cola queda en un estado válido.
- Es necesario también hacer CdE (controlar el valor de realloc o malloc), *antes* de asignar la nueva memoria a `q->data`.
- Un error menos frecuente, pero ilustrativo, es el de los que han intentado evitar el problema anterior copiando los datos a una nueva cola `qaux` antes de hacer crecer `q->data`. El problema con esta solución es que `qaux` será creada con su tamaño por defecto (`INIT_QUEUE_SIZE`), y por tanto no hay garantía de que `qaux` tenga espacio suficiente para todos los elementos -con lo cual habrá que hacer crecer `qaux` durante el proceso de copia, que se encontrará con el mismo problema, y así sin fin.

Es posible implementar una solución algo más eficiente que usa `realloc`, como puede verse a continuación. Pero téngase en cuenta que, en último término, ambas soluciones están copiando todos los datos a la nueva memoria reservada: `realloc` los copia también, aunque "no se vea".

```c
// SOLUCIÓN ALTERNATIVA, MÁS EFICIENTE
Status queue_grow(Queue *q) {
  void **new_data = NULL;
  int i, j, old_size, new_size;

  if (!q)
    return ERROR;

  old_size = q->capacity;
  new_size = q->capacity * QUEUE_GROWTH_FACTOR;

  // allocate new memory
  new_data = realloc(sizeof(void *) * new_size);

  if (!new_data)
    return ERROR;

  if (q->rear < q->front) {
    // The elements from q->data[0] to q->data[q->rear-1] need to be moved
       to
    // the positions that start at new_data[old_size] (right after the
       position
    // of the rightmost element of q->data) The number of data to be moved
       is
    // q->rear, so you move q->rear * sizeof(void*) bytes to the position
       that
    // starts at new_data+old_size
    // https://www.tutorialspoint.com/c_standard_library/c_function_memmove
       .htm
    memmove(new_data + old_size, new_data, q->rear * sizeof(void *));
    q->rear += old_size;
  }

  free(q->data);
  q->data = new_data;
  q->capacity = new_size;
  return OK;
}
```

b)

```c
Status queue_push(Queue *q, const void *elem) {
  Status st = OK;

  if (!q || !elem)
    return ERROR;

  if (q->front == (q->rear + 1) % (q->capacity)) {
    // queue is at full capacity
    st = queue_grow(q);
    if (st == ERROR)
      return ERROR;
  }
  q->data[q->rear] = (void *)elem;
  q->rear = (q->rear + 1) % q->capacity;
  return OK;
}
```

El error más común en esta parte fue usar `queue_is_full`. La cola en esta implementación nunca está llena, y en realidad la función solo la hemos definido con `MAX_QUEUE`, que no está definido aquí. Dicho esto, el test que usamos aquí es básicamente el mismo, pero usando `q->`

`capacity` en vez `MAX_QUEUE`.