

Examen Final

Algoritmos y Estructuras de Datos II - Taller

El examen consiste en implementar el TAD *Uno-Game* que representa una partida del juego de cartas llamado UNO. En realidad es una versión del juego con algunas modificaciones.



Figura 1: Un ejemplo de partida del juego de cartas UNO

La Baraja

Primeramente la baraja del juego consta de cartas con cuatro colores: rojo, verde, azul y amarillo con numeración del 0 a 9. Además están las cartas especiales *Saltar Turno* () y *Cambio de Color* (las cartas más divertidas +2, +4 y la de invertir el orden de la ronda se dejan afuera para simplificar). En la **figura 2** se muestran las cartas que se utilizan en esta versión del juego.



Figura 2: Cartas del juego UNO simplificado

Reglas del Juego

Aunque el TAD a implementar representa la pila de descarte (ver figura 1) y no se programará al juego completo, vale enunciar las reglas del juego en general. Inicialmente se reparten 7 cartas a cada jugador, se coloca el mazo boca abajo y se da vuelta una carta. La carta que se da vuelta es para iniciar la pila de descarte, donde los jugadores deben tirar una carta en su turno. Las reglas de la partida son:

1. Cada jugador está obligado a tirar una carta del color que se está jugando o que tenga un número igual al de la carta que se acaba de tirar a la pila de descarte en el turno anterior.
2. Si en su turno un jugador no tiene una carta compatible para tirar, debe sacar una a una cartas del mazo hasta conseguir alguna compatible. Una vez conseguida debe tirar esa carta.
3. Las cartas especiales *Saltar Turno* () sólo se pueden tirar contra una carta del mismo color o si en el tope de la pila de descarte hay también una carta *Saltar Turno* (sin importar su color). Indica que quien nos sigue pierde el turno.
4. Las cartas especiales *Cambio de Color* se pueden tirar sin importar qué color se tiro previamente. Si se tira, se debe elegir el color que se seguirá jugando.
5. Si al repartir las cartas en el comienzo de una partida la primer carta que se da vuelta es una *Cambio de Color*, ese es el único caso en el que la carta se pone en el mazo y se vuelve a barajar. Es decir, la pila de descarte no puede comenzar con una carta *Cambio de Color*.
6. Gana aquel que se queda sin cartas, pero solo si al quedarse con una carta en sus manos grita «uno» de manera bien audible. Si no lo hace, los compañeros le reclamarán no haber dicho «uno» y debe llevarse una carta de penalización del mazo.

Notar que en esta versión del juego **en cada turno siempre se tira una carta a la pila de descarte**.

TAD *Card* [card.h, card.c]

El primer tipo abstracto de datos que se debe implementar es el TAD *Card*, que representa una carta del juego. Para ello se cuenta con los tipos enumerados `color_t` y `type_t` que representan los colores y los tipos de carta respectivamente:

```
typedef enum {red, green, blue, yellow} color_t;
typedef enum {normal, change_color, skip} type_t;
```

La interfaz del TAD *Card* está dada por las siguientes funciones:

Función	Descripción
<code>card_t card_new(unsigned int n, color_t c, type_t t)</code>	Crea una nueva instancia del TAD usando memoria dinámica, que representa una carta con numeración <code>n</code> , color <code>c</code> y tipo <code>t</code>
<code>unsigned int card_get_number(card_t card)</code>	Devuelve la numeración de la carta <code>card</code>
<code>color_t card_get_color(card_t card)</code>	Devuelve el color de la carta <code>card</code>
<code>type_t card_get_type(card_t card)</code>	Devuelve el tipo de la carta <code>card</code>
<code>bool card_samenum(card_t c1, card_t c2)</code>	Indica si <code>c1</code> y <code>c2</code> tienen el mismo número
<code>bool card_samecolor(card_t c1, card_t c2)</code>	Indica si las cartas <code>c1</code> y <code>c2</code> tienen el mismo color
<code>bool card_compatible(card_t new_card, card_t pile_card)</code>	Indica si la carta <code>new_card</code> es compatible con la carta <code>pile_card</code> , suponiendo que en el juego <code>pile_card</code> está al tope de la pila de descarte y se quiere tirar la carta <code>new_card</code> .
<code>card_t card_destroy(card_t card)</code>	Destruye la instancia <code>card</code> liberando toda la memoria utilizada por ella

Representación

Internamente el TAD Card se representa con la siguiente estructura:







```
struct s_card {
    unsigned int num;
    color_t color;
    type_t type;
};
```

Para referirnos a los valores de la estructura se va a seguir la siguiente notación:

```
[<num><color>:<tipo>]

<num>    ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<color>   ::= r | g | b | y
<tipo>    ::= n | c | s
```

Algunos ejemplos de la notación:

Notación	[4r:n]	[0b:n]	[0r:s]	[0b:s]	[0b:c]	[0g:c]
Carta						
Descripción	Carta 4 roja	Carta 1 azul	Carta Saltear Turno roja	Carta Saltear Turno azul	Carta Cambio de Color - Establece el color azul	Carta Cambio de Color - Establece el color verde

Notar que para las cartas especiales (cuyo tipo no es `normal`) la numeración es siempre 0 puesto que para ese tipo de cartas no tiene sentido hablar de numeración.





Compatibilidad

La función `card_compatible()` considera que una carta `new_card` es compatible con una carta `pile_card` si un jugador tiene permitido, según las reglas del juego, tirar la carta `new_card` cuando en la pila de descarte está la carta `pile_card`.

Alguno ejemplos donde `card_compatible(new_card, pile_card)` devuelve `true`, es decir, donde `new_card` es compatible con `pile_card` son:

new_card	 [7g:n]	 [4r:n]	 [4g:n]	 [0b:s]	 [0r:s]
pile_card	 [4g:n]	 [4g:n]	 [4g:n]	 [3b:n]	 [0y:s]
Razón	Mismo color	Mismo número	Mismo número y color	Mismo color	Ambas Saltear Turno

Ejemplos usando la carta de *Cambio de Color*:

new_card	 [0g:c]	 [0r:c]	 [0y:c]	 [7b:n]	 [0r:s]
pile_card	 [3b:n]	 [4g:n]	 [0b:c]	 [0b:c]	 [0r:c]
Razón	(1)	(2)	(3)	(4)	(5)

Los casos (1), (2) y (3) son compatibles ya que las cartas especiales *Cambio de Color* siempre se pueden tirar, sin importar qué carta esté en la pila de descarte (como indica la *regla 4* del juego). Las cartas de los casos (4) y (5) son compatibles porque en el tope de la pila, la carta *Cambio de Color* estableció como color de juego al mismo color de la carta que se quiere tirar.

Algunos ejemplos incompatibles, es decir donde `card_compatible(new_card, pile_card)` debe dar `false` como resultado son:

new_card	 [1r:n]	 [0g:n]	 [2y:n]	 [0r:s]
pile_card	 [4g:n]	 [0r:s]	 [0b:c]	 [0b:c]

TAD *Uno-Game* [unogame.h, unogame.c]

Este TAD guardará la información contenida en la pila de descarte del juego. Es en esencia una pila, con elementos del tipo `card_t` y además guarda qué jugador tiró la carta en cuestión. Un jugador se representa con un número del tipo `player_t` definido como:

```
typedef int player_t;
```

La estructura de nodos entonces debe guardar ambos datos, **la carta y el jugador que la jugó**. Los jugadores se numeran desde el cero. Notar que al principio, la pila de descarte tiene sólo una carta, la cual nadie tiró sino que se sacó del mazo para empezar la partida. Entonces hay que usar un valor especial para asociar a esa primera carta. Se recomienda usar `-1`, que representará que esa carta no la jugó ningún jugador. Esto sólo pasará con la primera carta de la pila, todas las demás deben tener asociado un valor de `player_t` mayor o igual a cero. La cantidad de jugadores que participan del juego está dada por la constante `NUM_PLAYERS` que está definida en `unogame.h` y por defecto vale 3 pero debe funcionar para cualquier valor al que se le de a `NUM_PLAYERS`.

La interfaz del TAD está dada por las siguientes funciones:

Función	Descripción
<code>unogame_t uno_newgame(card_t card)</code>	Construye una nueva instancia del TAD donde la pila de descarte tiene a <code>card</code> como carta inicial. La carta inicial no se chequea.
<code>card_t uno_topcard(unogame_t uno)</code>	Devuelve la carta que está en el tope de la pila de descarte en la instancia del TAD <code>uno</code>
<code>unogame_t uno_addcard(unogame_t uno, card_t c)</code>	Agrega una carta a la pila de descarte de la instancia <code>uno</code> . La función no chequea compatibilidad entre <code>c</code> y el tope de la pila.
<code>unsigned int uno_count(unogame_t uno)</code>	Devuelve la cantidad de cartas que hay en la pila de descarte de la instancia <code>uno</code>
<code>color_t uno_currentcolor(unogame_t uno)</code>	Devuelve el color vigente del juego, es decir el color de la carta del tope de la pila de descarte de la instancia <code>uno</code>
<code>player_t uno_nextplayer(unogame_t uno)</code>	Según la carta que está al tope de la pila de descarte y el jugador que la tiró, indica el número de jugador del próximo turno.
<code>bool uno_validpile(unogame_t uno)</code>	Indica si la pila está bien formada, respetando las reglas del juego.
<code>card_t *uno_pile_to_array(unogame_t uno)</code>	Devuelve un arreglo en memoria dinámica con las cartas en el orden que se jugaron.
<code>unogame_t uno_destroy(unogame_t uno)</code>	Destruye la instancia <code>uno</code> , destruyendo la pila de descarte y todas las cartas contenidas en ella.

Representación

Para representar la estructura de nodos donde se guarda la pila de descarte vamos a usar la notación:

```
[2b:n] 1 -> [4b:n] 0 -> [4r:n] 2 -> [7r:n] 1 -> [7g:n] 0 -> [3g:n] -1 -> NULL
```

Los nodos del ejemplo representan la pila de la *figura 3a*, donde en el tope hay una carta azul con numeración 2, jugada por el jugador número 1 (el segundo jugador). La función `uno_addcard()` debe entonces además de agregar una carta a la pila, asignarle el jugador correspondiente. Esto es casi trivial, pero se debe tener en cuenta la carta de salto de turno ya que para la pila de la *figura 3b* la estructura sería:

```
[1r:n] 1 -> [1g:n] 0 -> [0g:s] 1 -> [8g:n] 0 -> [8a:n] -1 -> NULL
```

Una buena idea es delegar el problema de determinar quién es el próximo jugador a la función `uno_nextplayer()` y utilizarla de manera auxiliar en `uno_addcard()`.

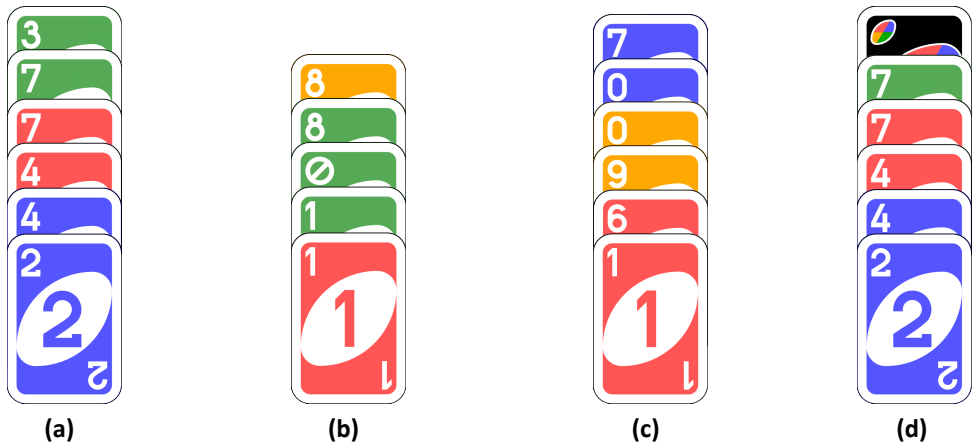


Figura 3: Ejemplos de pilas de descarte

Pilas Válidas

Lo que no hace `uno_addcard()` es verificar si la carta se juega correctamente, por ello podría darse la situación de la **figura 3c**. En ese caso la función `uno_validpile()` devuelve `false`. Algo similar pasa con `uno_newgame()` que permite construir una pila como la de la **figura 3d**, donde se comenzó la pila con una carta de *Cambio de Color* lo cual no está permitido por las reglas. A continuación se muestran ejemplos del funcionamiento de `uno_validpile()`:

Pila de descarte	Retorno	Razón
[4y:n] 0 -> [4r:n] 2 -> [0r:s] 0 -> [7r:n] -1 -> NULL	true	Todas las cartas son compatibles con su anterior
[9b:n] 2 -> [0b:c] 1 -> [7g:n] 0 -> [7r:n] -1 -> NULL	true	Todas las cartas son compatibles con su anterior. Notar que hay una carta de cambio de color [0b:c], que establece al azul como color de juego, a la cual luego se le tira una carta azul [9b:n].
[9b:n] 2 -> [0b:c] 1 -> [6b:n] 0 -> [7r:n] -1 -> NULL	false	La carta [6b:n] no es compatible con [7r:n]
[3y:n] 2 -> [0b:c] 1 -> [7g:n] 0 -> [7r:n] -1 -> NULL	false	La carta [3y:n] no es compatible con [0b:c] ya que esta última cambio al color azul pero luego se tira una carta de color amarillo.
[4g:n] 0 -> [0g:n] 2 -> [0r:s] 0 -> [7r:n] -1 -> NULL	false	Las cartas [0g:n] y [0r:s] no son compatibles ya que la una es la carta 0 de color verde y la otra es la carta de salto de turno con color distinto. Aunque en la estructura tengan el mismo número son de distinto tipo.
[9b:n] 2 -> [0b:c] 1 -> [7g:n] 0 -> [0g:c] -1 -> NULL	false	La pila comienza con una carta de <i>Cambio de Color</i> lo cual está prohibido por la regla 5 del juego.

Notar que en el chequeo de validez que se pide, no hace falta tener en cuenta el número de jugador que juega cada carta, sino la compatibilidad de las mismas.

Arreglos

La función `uno_pile_to_array()` debe devolver un arreglo dinámico con las cartas de la pila de descarte en el orden en que fueron agregadas. La cantidad de elementos contenidos en el arreglo se debe corresponder con el valor devuelto por `uno_count()`. Por ejemplo para la pila:

```
[2b:n] 1 -> [4b:n] 0 -> [4r:n] 2 -> [7r:n] 1 -> [7g:n] 0 -> [3g:n] -1 -> NULL
```

Se debe devolver el arreglo:

```
[ [3g:n], [7g:n], [7r:n], [4r:n], [4b:n], [2b:n] ]
```

Compilación y Test

Se provee un `Makefile` para compilar todo el código y generar un ejecutable. Para ello deben hacer:

```
$ make
```

y luego pueden probar su implementación con los archivos de ejemplo de la carpeta `input`. Los ejemplos incluidos en `input` asumen que la cantidad de jugadores es `3` por lo que si se utilizan los ejemplos con otro valor de `NUM_PLAYERS` los resultados no serán correctos. Para correr un ejemplo de un archivo en particular se debe ejecutar:

```
$ ./test_uno -f input/example_01.in
```

Si todo sale bien se debería obtener la siguiente salida:

```
READING input/example_01.in
Reading PILE from input/example_01.in...
input      : [4y:n] 0 -> [4r:n] 2 -> [0r:s] 0 -> [7r:n] -1 -> NULL
building pile : [4y:n] 0 -> [4r:n] 2 -> [0r:s] 0 -> [7r:n] -1 -> NULL
length reported: 4
next player  : 1
valid pile   : true
array        : [ [7r:n], [0r:s], [4r:n], [4y:n] ]
DONE input/example_01.in.
```

El ítem `input` se refiere a la pila guardada en el archivo de ejemplo, mientras que `building pile` se refiere a la pila que se construyó usando el TAD a partir de las entradas del archivo. Si la implementación es correcta se verá la misma pila en ambos ítems. El ítem `length reported` es la longitud de la pila según la implementación del TAD. El ítem `next player` es el resultado de la función `uno_nextplayer()` para la pila del ejemplo y el ítem `valid pile` indica si el TAD considera a la pila válida (es decir, el resultado de la función `uno_valid_pile()`)

Está disponible la opción de verificación para comparar los resultados de sus funciones con los valores esperados para el ejemplo. Para ello:

```
$ ./test_uno -vf input/example_01.in

READING input/example_01.in
Reading PILE from input/example_01.in...
input      : [4y:n] 0 -> [4r:n] 2 -> [0r:s] 0 -> [7r:n] -1 -> NULL
building pile : [4y:n] 0 -> [4r:n] 2 -> [0r:s] 0 -> [7r:n] -1 -> NULL [OK]
length reported: 4 [OK]
next player  : 1 [OK]
valid pile   : true [OK]
array        : [ [7r:n], [0r:s], [4r:n], [4y:n] ] [OK]
DONE input/example_01.in.
```

En caso de error se muestra el valor esperado y además se muestra la cantidad de errores ocurridos. Se puede ejecutar además usando múltiples archivos al mismo tiempo:

```
$ ./test_uno -vf input/example_01.in input/example_02.in
```

obteniendo la siguiente salida (si la implementación no falla):

```
READING input/example_01.in
Reading PILE from input/example_01.in...
input      : [4y:n] 0 -> [4r:n] 2 -> [0r:s] 0 -> [7r:n] -1 -> NULL
building pile : [4y:n] 0 -> [4r:n] 2 -> [0r:s] 0 -> [7r:n] -1 -> NULL [OK]
length reported: 4 [OK]
next player  : 1 [OK]
valid pile   : true [OK]
array        : [ [7r:n], [0r:s], [4r:n], [4y:n] ] [OK]
DONE input/example_01.in.

READING input/example_02.in
Reading PILE from input/example_02.in...
input      : [9b:n] 2 -> [0b:c] 1 -> [7g:n] 0 -> [7r:n] -1 -> NULL
building pile : [9b:n] 2 -> [0b:c] 1 -> [7g:n] 0 -> [7r:n] -1 -> NULL [OK]
length reported: 4 [OK]
next player  : 0 [OK]
valid pile   : true [OK]
array        : [ [7r:n], [7g:n], [0b:c], [9b:n] ] [OK]
DONE input/example_02.in.

ALL TESTS OK
```

También se pueden usar los comodines y correr los test para todos los archivos que comienzan con “example_”:

```
$ ./test_uno -vf input/example_*
```

Para compilar y realizar tests usando todos los ejemplos de la carpeta **input** en modo verificación se puede hacer directamente:

```
$ make test
```

Para además chequear con valgrind:

```
$ make valgrind
```

IMPORTANTE: Pasar los tests no significa aprobar. Tener *memory leaks* resta puntos.

Desactivar visualizado de colores

En caso de usar Windows u otro sistema no compatible con los colores de consola, para desactivar los colores deben editar el archivo `pretty_helpers.c` y descomentar la siguiente línea:

```
#include <stdio.h>

// Para desactivar los colores deben descomentar la siguiente línea:
// #define NO_PRETTY <=====

// Color ANSI code
(:)
```

de manera que quede:

```
#include <stdio.h>

// Para desactivar los colores deben descomentar la siguiente línea:
#define NO_PRETTY

// Color ANSI code
(:)
```

una vez hecho esto se puede compilar y ejecutar sin problemas.