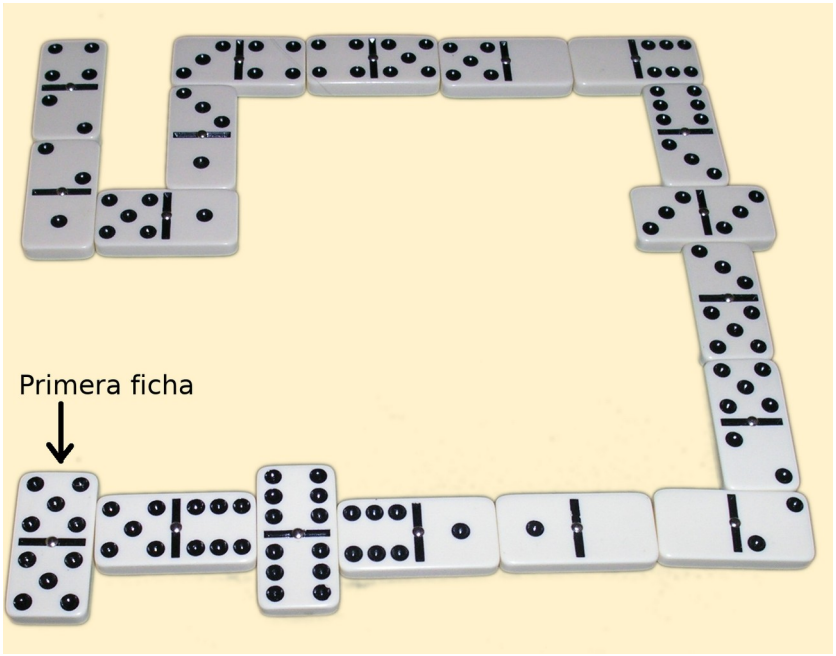
Córdoba, 06 de Febrero del 2024

Examen Final

Algoritmos y Estructuras de Datos II - Taller

El ejercicio consiste en implementar el TAD *Domino-Line* que representa una línea de juego correspondiente a una partida de Dominó.



Ejemplo de una línea de juego de dominó

Como se ve en la figura, la línea de juego es una secuencia de fichas (*tiles*) por lo que para implementar el TAD **debe usarse una lista enlazada** de fichas de dominó. Una ficha de dominó se representa con el TAD *Domino* que **está mayormente implementado** y sólo deben completar un par de sus funciones.

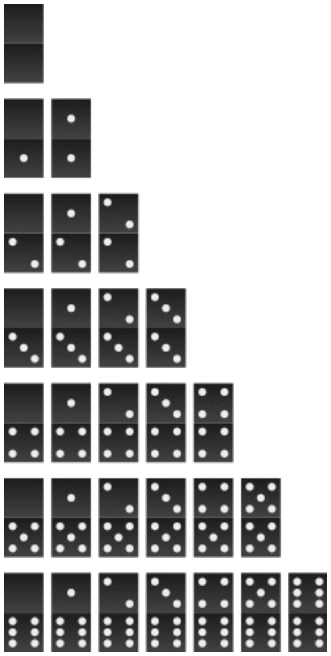
TAD *Domino*

El juego Dominó cuenta con 28 fichas rectangulares donde en cada una de ellas figuran dos números. Para referirnos a los números vamos a pensar que están orientadas verticalmente:

(a)

(b)

Entonces, **(a)** es la ficha con un **dos** arriba y un **cuatro** abajo y **(b)** la ficha con **cero** arriba y un **cinco** abajo. Los números pueden ser 0, 1, 2, 3, 4, 5 o 6. Para referirnos a una ficha usaremos la notación **`n:m`**. Entonces la ficha (a) se puede escribir como **`2:4`** y la ficha (b) **`0:5`**.

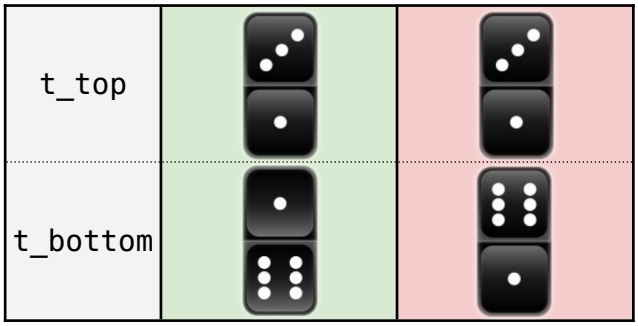


El TAD *Domino* tiene la siguiente interfaz:

Función	Descripción
`domino domino_new(int num_up, int num_down)` `-- completar --`	Crea una ficha con numeración `num_up` arriba y con numeración `num_down` abajo
`bool domino_is_double(domino tile)`	Indica si la ficha es de la forma **`n:n`**
`int domino_up(domino tile)`	Devuelve el número superior de la ficha
`int domino_down(domino tile)`	Devuelve el número inferior de la ficha
`bool domino_matches(domino t_top, domino t_botton)`	Indica si la ficha `t_top` encaja con la ficha `t_bottom`.
`void domino_dump(domino tile)`	Muestra una ficha por pantalla
`domino domino_destroy(domino tile)` `-- completar ---`	Destruye una instancia del TAD *Domino*, liberando toda la memoria utilizada

Encaje de fichas

La función `domino_matches()` considera que una ficha `t_top` *encaja* con una ficha `t_bottom` si al colocar `t_top` por encima de `t_bottom` los números que se "tocan" coinciden, en caso contrario no encajan.



En otras palabras, el número de abajo de `t_top` debe coincidir con el número de arriba de `t_bottom`.

TAD *Domino-Line*

Las funciones a implementar en **`domino_line.c`** son las siguientes:

Función	Descripción
`domino_line line_new(domino first)`	Construye una línea de juego con la ficha inicial `first`.
`domino_line line_add(domino_line line, domino t)`	Agregar una ficha a la línea de juego
`unsigned int line_length(domino_line line)`	Devuelve la longitud de la línea de juego
`bool line_n_correct(domino_line line, unsigned int n)`	¿La n-ésima ficha está bien ubicada?
`int line_total_points(domino_line line)`	Devuelve la suma de todas las fichas
`domino * line_to_array(domino_line line)`	Arreglo dinámico con las fichas de la línea
`void line_dump(domino_line line)`	Muestra la línea de juego en la pantalla
`domino_line line_destroy(domino_line line)`	Destruye la línea de juego y todas las fichas

Fichas bien ubicadas - `line_n_correct()`

Se considera que una ficha `t` está bien ubicada cuando la ficha anterior en la línea de juego encaja con `t` y además `t` encaja con la ficha que le sigue. Para la primera ficha de la línea, se considera bien ubicada si ésta encaja con la segunda ficha. La última ficha está bien ubicada si la penúltima

encaja con ella. Si la línea tiene una sola ficha siempre estará bien ubicada. Las posiciones se cuentan desde cero, por lo que **`line_n_correct(line, 1)`** indica si la segunda ficha está ubicada correctamente. Se puede entender mejor con los siguientes ejemplos:

Línea de Juego	*n*	*Retorno*	*Razón*
[4:4 4:1 1:6 3:6]	1	true	La ficha 4:4 encaja con 4:1 y 4:1 encaja con 1:6
[4:4 4:1 1:6 3:6]	2	false	La ficha 4:1 encaja con 1:6 pero 1:6 **no encaja** con 3:6
[4:4 4:1 1:6 3:6]	3	false	No está bien ubicada ya que 1:6 **no encaja** con 3:6
[4:4 4:1 1:6 3:6]	0	true	La ficha 4:4 esta bien ubicada ya que encaja con 4:1
[2:2]	0	true	Como es la única ficha, está trivialmente bien ubicada

TIP: Puede ser buena idea separar los casos **`n==0`** y **`n==line_length(line)-1`** ya que en los demás casos intervienen 3 fichas (la anterior, la actual y la siguiente).

Suma de fichas

Dependiendo de la variante de Dominó en la que se juega, la suma de los puntos de las fichas de la línea de juego determinan el puntaje del equipo ganador. Una ficha suma tantos puntos como la suma de los dos números que aparecen en ella, es decir que una ficha **`n:m`** tiene **`n + m`** puntos. La función `line_total_points()` suma los puntos de todas las fichas en la línea de juego. Por ejemplo:

Línea de Juego	*Retorno*
[4:4 4:1 1:6 3:6]	(4+4) + (4+1) + (1+6) + (3+6) = 29
[6:6 1:3 0:1 0:0]	(6+6) + (1+3) + (0+1) + (0+0) = 17
[3:3 3:1 1:2 2:2]	(3+3) + (3+1) + (1+2) + (2+2) = 17

Arreglos

La función `line_to_array()` debe devolver un arreglo dinámico con las fichas de dominó de la línea de juego en el orden en que fueron agregadas. La cantidad de elementos contenidos en el arreglo se debe corresponder con el valor devuelto por `line_length()`.

Iterator

En numerosos lenguajes, una conocida técnica para recorrer una estructura contenedora de elementos es la llamada “iterator”. Consiste en proveer un conjunto de funciones que son de ayuda para no acceder a la representación interna del tipo abstracto de datos pero permiten “iterar” los elementos internos. De esta manera se pueden desarrollar funciones complementarias sin tener la necesidad de modificar el TAD. Implementar las siguienes funciones

Función	Descripción
line_iterator line_iterator_begin**(domino_line** line**)**	Construye un nuevo iterador a line y lo inicializa apuntando al primer elemento
Domino line_iterator_get**(line_iterator** lit**)**	Devuelve el elemento Domino que esta en la posición actual del iterador. PRECONDICION: El iterador debe apuntar a un elemento válido de la lista (line_iterator_end(lit) != NULL)
line_iterator line_iterator_next**(line_iterator** lit**)**	Avanza el iterador a la siguiente posición disponible (O NULL si se ha llegado ya al final de la lista). Esta función es idempotente si ya se ha alcanzado el final, es decir line_iterator_next(NULL) == NULL
bool line_iterator_end**(line_iterator** lit**)**	Devuelve “true” si ya se ha alcanzado el final de la lista, “false” en caso contrario. Por final de la lista entendemos que ya no hay elementos por procesar. Es decir para la lista [1:2, 2:2, 3:0] si el iterador “lit” está apuntado a 3:0 esta función devolverá **FALSE,** pues existe un elemento todavía por procesar.

Funciones complementarias

Gracias a tener implementado el concepto de iteradores, sin violar la abstracción del TAD podemos crear funciones complementarias. Implementar en el archivo main.c las siguientes funciones:

Función	Descripción
unsigned int doubles_count**(domino_line** line**)**	Devuelve la cantidad de piezas dobles que hay en line.
bool is_in_line **(domino_line** line, **domino** tile**)**	Devuelve true si la pieza “tile” está presente en line, false en caso contrario.

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 implemetación con los archivos de ejemplo de la carpeta **input**:

```
$ ./test_line -f input/example01.in
```

si todo sale bien debería obtener la siguiente salida:

```
READING input/example01.in
Reading TILES from file...
Building domino-line: [ 4:4, 4:1, 1:6, 3:6]
length reported: 4
check correct tiles: [ 4:4 (T), 4:1 (T), 1:6 (F), 3:6 (F) ]
total points: 29
array: [ 4:4, 4:1, 1:6, 3:6 ]
DONE input/example01.in.
```

además pueden usar la opción de verificación para comparar los resultados de sus funciones con los valores esperados para el ejemplo. Para ello:

```
$ ./test_line -vf input/example01.in
```

La salida obtenida:

```
READING input/example01.in
Reading TILES from file...
Building domino-line: [ 4:4, 4:1, 1:6, 3:6]
length reported: 4 [OK]
check correct tiles: [ 4:4 (T), 4:1 (T), 1:6 (F), 3:6 (F) ] [OK]
total points: 29 [OK]
array: [ 4:4, 4:1, 1:6, 3:6 ] [OK]
DONE input/example01.in.

 ALL TESTS OK
```

En caso de error se muestra el valor esperado y además se muestra la cantidad de errores ocurridos. Si se omite la opción **`-f`** se lee la línea de juego por la entrada estándar, debiendo ingresar primero la cantidad de elementos y luego las fichas usando la notación **`n:m`** separadas por espacios o apretando enter:

```
READING stdin
Reading TILES from stdin...
3
1:1
2:2
3:3
Building domino-line: [ 1:1, 2:2, 3:3, 4:4, 5:5]
length reported: 5
check correct tiles: [ 1:1 (F), 2:2 (F), 3:3 (F), 4:4 (F), 5:5 (F) ]
total points: 30
array: [ 1:1, 2:2, 3:3, 4:4, 5:5 ]
DONE stdin.
```

Para realizar test usando todos los ejemplos de la carpeta **`input`** en modo verificación:

```
$ make test
```

Para además chequear con valgrind:

```
$ make valgrind
```

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