

# Examen Final de Laboratorio - Libres

## Algoritmos y Estructura de Datos II



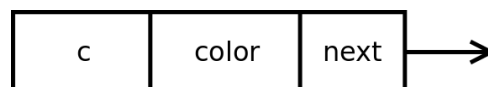
Se va a implementar un Tipo Abstracto de Datos que representa un mazo de cartas de poker. El mazo tiene la particularidad de que agrupa las cartas del mismo color.

Vamos a utilizar además un TAD Card que representará una carta del mazo.

El TAD *Card* tiene la siguiente interfaz:

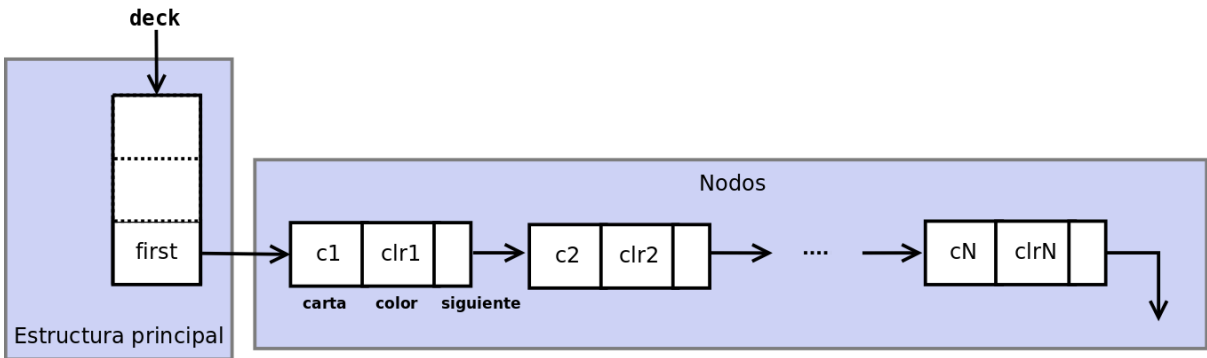
Función	Descripción
<code>card card_create(cardnum_t num, cardsuit_t suit)</code>	Crea una carta con numeración <code>num</code> y palo <code>suit</code> .
<code>cardnum_t card_number(card c)</code>	Retorna el número de la carta <code>c</code>
<code>cardsuit_t card_suit(card c)</code>	Retorna el palo de la carta <code>c</code>
<code>cardcolor_t card_color(card c)</code>	Retorna el color de la carta
<code>bool card_equals(card c1, card c2)</code>	Indica si las cartas <code>c1</code> y <code>c2</code> tienen el mismo número y palo
<code>card card_destroy(card c)</code>	Destruye una instancia del TAD <i>Card</i> , liberando toda la memoria utilizada

El TAD principal *SortedDeck* guarda las cartas en nodos simplemente enlazados. Cada nodo almacena una instancia del TAD *Card*, el color de la carta y un puntero al siguiente nodo:



Notar que el color de la carta `c` que está guardada en el nodo podría obtenerse llamando a `card_color(c)`, sin embargo realizar tantas llamadas a esa función no es eficiente y por ello los nodos deben guardar el color de la carta en el campo `color` cuando son creados.

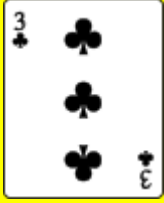
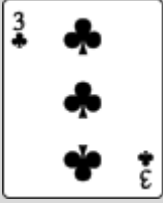
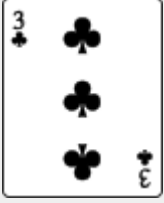

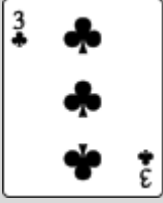

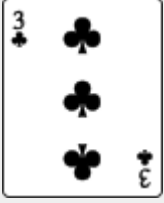

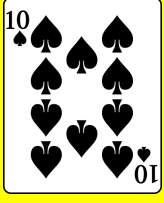
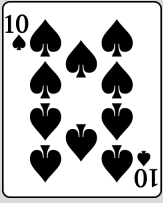
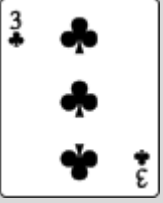

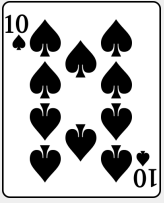
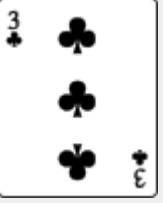


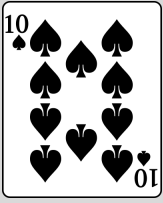
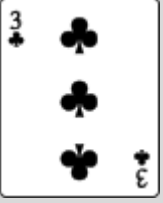


El TAD además debe implementarse con una estructura principal que tiene entre otros campos un puntero al primer nodo. El tipo definido para el TAD se llama **sdeck** y a continuación se muestra un esquema de una instancia del TAD apuntada por una variable **deck**:



Como se mencionó anteriormente, el mazo debe agrupar las cartas del mismo color. Entonces un ejemplo de cómo funciona el mazo puede verse a continuación:

Mazo actual	Agregar	Resultado
<mazo vacío>		

Siguiendo esa mecánica el mazo siempre tendrá dos secciones, una donde todas las cartas son rojas y otra donde todas las cartas son negras. **Esa propiedad fundamental debe mantenerse durante toda la vida de la instancia del TAD.** Aunque en este ejemplo la sección de las rojas va al principio, eso va depender de la primera carta ya que en este otro ejemplo:

Mazo actual	Agregar	Resultado
<mazo vacío>		
		 
 		  
  		   

Notar que ahora la sección de las negras está al principio.

La mecánica de los ejemplos permite garantizar la propiedad fundamental del TAD, sin embargo **cualquier implementación que logre mantener esta propiedad será tomada como correcta**, siempre y cuando se use la representación indicada en el esquema.

Las operaciones del TAD se listan a continuación:

Función	Descripción
<code>sdeck sorteddeck_create(void)</code>	Crea un nuevo mazo vacío
<code>bool sorteddeck_is_empty(sdeck deck)</code>	Indica si el mazo <code>deck</code> es vacío o no
<code>sdeck sorteddeck_add(sdeck deck, card c)</code>	Agrega a <code>deck</code> una carta <code>c</code> . El TAD se apropia de la instancia de esa carta, por lo que <b>es responsabilidad del TAD SortedDeck destruirla cuando ya no se utilice</b> .
<code>sdeck sorteddeck_remove(sdeck deck, card c)</code>	Elimina del mazo <code>deck</code> una carta que tenga la misma numeración y palo que la carta <code>c</code> . Si en <code>deck</code> no hay ninguna carta igual a <code>c</code> , el mazo queda sin modificaciones. Si hubiera más de una carta igual a <code>c</code> , elimina alguna (sólo una) de ellas.
<code>sdeck sorteddeck_popfirst(sdeck deck)</code>	Elimina la carta que está al principio del mazo <code>deck</code>

<code>card sorteddeck_first(sdeck deck)</code>	Devuelve la carta que está al principio del mazo <b>deck</b> . Devuelve la instancia interna del almacenada en el TAD, <b>no una copia</b> .
<code>unsigned int sorteddeck_size(sdeck deck)</code>	Indica la cantidad de cartas que hay en <b>deck</b> . Debe tener orden constante $O(1)$ .
<code>unsigned int sorteddeck_redcount(sdeck deck)</code>	Cuenta la cantidad de cartas de color rojo en <b>deck</b> . Debe tener orden constante $O(1)$ .
<code>unsigned int sorteddeck_blackcount(sdeck deck)</code>	Cuenta la cantidad de cartas de color negro en <b>deck</b> . Debe tener orden constante $O(1)$ .
<code>card* sorteddeck_to_array(sdeck deck);</code>	Devuelve un arreglo en memoria dinámica que contiene todas las cartas de <b>deck</b> ordenadas tal como se encuentran almacenadas en los nodos. Las cartas almacenadas en el arreglo <b>deben ser copias</b> de las cartas que se encuentran en <b>deck</b> .
<code>void sorteddeck_dump(sdeck deck)</code>	Muestra todas las cartas del mazo <b>deck</b> en el orden que aparecen en los nodos.
<code>sdeck sorteddeck_destroy(sdeck deck)</code>	Destruye el mazo <b>deck</b> liberando toda la memoria utilizada por la instancia.

El programa resultante no debe dejar *memory leaks* ni lecturas/escrituras inválidas.

Se debe definir una *invariante de representación*, aunque sea una básica pero no trivial. La invariante también debe verificarse en las pre y post condiciones que correspondan.

Se provee un módulo `testing.c` que implementa una interfaz para poder probar las funciones del TAD. Además se incluye un `Makefile`. Una vez compilado el programa puede probarse ejecutando:

```
$ ./testdeck
```

También se puede hacer directamente

```
$ make test
```

Consideraciones:

- Solo se deben modificar los archivos `card.c` y `sorteddeck.c`
- Se incluyen un par de funciones auxiliares `create_node()` y `destroy_node()` que son las que se deben encargarse de crear y destruir nodos. Esto ayuda a no duplicar código.
- Se provee el archivo `Makefile` para facilitar la compilación.
- Se recomienda usar las herramientas `valgrind` y `gdb`.
- Si el programa no compila, no se aprueba el examen.
- Los *memory leaks* bajan puntos
- Entregar código muy impropio puede restar puntos
- Si `sorteddeck_size()`, `sorteddeck_redcount()` y `sorteddeck_blackcount()` no son de orden constante  $O(1)$  baja muchísimos puntos.
- **No modificar los .h.**