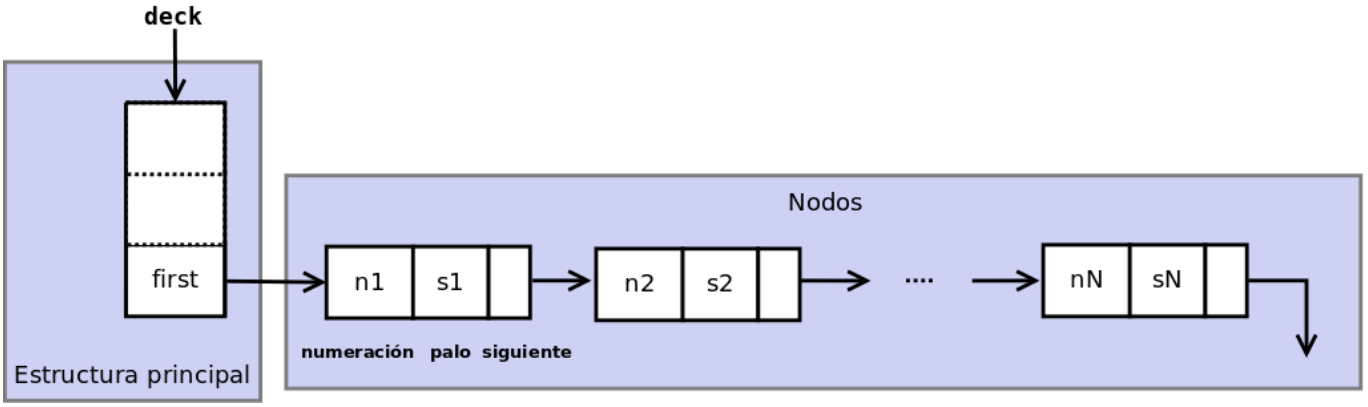
Algoritmos y Estructuras de Datos II

Parcial 31-05: Tema C - TAD: Mazo de Poker

Se va a implementar un Tipo Abstracto de Datos que representa un mazo de cartas de poker. Las cartas son guardadas en nodos simplemente enlazados, que contienen la numeración y palo de la carta y un puntero al siguiente nodo. El TAD además debe implementarse con una estructura principal que tiene entre otros campos un puntero al primer nodo. El tipo se llama **`pokerdeck`** y si se tiene una variable **`deck`** de ese tipo la representación se puede ver en el siguiente esquema:



Cuando el mazo está vacío, el puntero **`first`** de la estructura principal apuntará a **`NULL`**. Dentro de las definiciones del TAD están además los tipos **`cardnum_t`** y **`cardsuit_t`** que representan las numeraciones de cartas y los palos respectivamente. O sea que **`n1`** es de tipo **`cardnum_t`** y **`s1`** del tipo **`cardsuit_t`** (obviamente **`n2`**, ..., **`nN`** tienen el mismo tipo que **`n1`** y lo mismo con **`s2`**, ..., **`sN`** que tienen el mismo tipo que **`s2`**). Particularmente el tipo **`cardsuit_t`** es un tipo enumerado por lo que los valores de **`s1`**, **`s2`**, ..., **`sN`** deberían ser algunas de las constantes definidas. Por otro lado **`cardnum_t`** no es enumerado sin embargo se definen constantes que determinan su rango válido.

Las operaciones del TAD se listan a continuación:

Función	Descripción
`pokerdeck pokerdeck_empty(void)`	Crea un nuevo mazo vacío
`bool pokerdeck_is_empty(pokerdeck deck)`	Indica si el mazo **deck** es vacío o no
`pokerdeck pokerdeck_add(pokerdeck deck, cardnum_t num, cardsuit_t suit)`	Agrega a **deck** una carta con numeración **num** y palo **suit** ubicándola en el fondo del mazo. Si luego de agregar la carta con esta función se llama inmediatamente a **pokerdeck_pop()** la carta recién agregada no se eliminaría salvo que el mazo sólo tuviera esa carta.
`pokerdeck pokerdeck_push(pokerdeck deck, cardnum_t num, cardsuit_t suit)`	Agrega a **deck** una carta con numeración **num** y palo **suit** ubicándola en el tope del mazo. Si luego de agregar la carta con esta función se llama inmediatamente a **pokerdeck_pop()** se eliminaría la carta que se acaba de agregar.
`pokerdeck pokerdeck_pop(pokerdeck deck, cardnum_t *popped_num, cardsuit_t *popped_suit)`	Elimina la carta que está al tope del mazo **deck**. La numeración de la carta eliminada se almacena en la memoria apuntada por **popped_num** mientras que el palo de la carta eliminada se almacena en la memoria apuntada por **popped_suit**. Si no se desea almacenar alguno de estos valores se puede pasar **NULL** a **popped_num** o **popped_suit**.
`unsigned int pokerdeck_length(pokerdeck deck)`	Devuelve la cantidad de cartas que hay en **deck**.
`pokerdeck pokerdeck_remove(pokerdeck deck, cardnum_t num, cardsuit_t suit)`	Elimina una carta con numeración **num** y palo **suit** del mazo **deck**. Si hubiera más de una carta con esa numeración y palo elimina alguna (sólo una) de ellas.
`unsigned int pokerdeck_count(pokerdeck deck, cardsuit_t suit)`	Cuenta la cantidad de cartas que hay en **deck** que son del palo **suit**.
`struct card * pokerdeck_to_array(pokerdeck deck)`	Devuelve un arreglo en memoria dinámica que contiene todas las cartas de **deck** ordenadas de tal manera que el tope del mazo se ubica en la primera posición del arreglo y en la última posición se ubica la carta del fondo del mazo. Los elementos son del tipo **struct card**.
`void card_dump(cardnum_t num, cardsuit_t suit)`	Muestra una carta con numeración **num** y palo **suit** por pantalla
`void pokerdeck_dump(pokerdeck deck)`	Muestra todas las cartas del mazo **deck** en orden desde el tope hasta el fondo del mazo.
`pokerdeck pokerdeck_destroy(pokerdeck deck)`	Destruye el mazo **deck** liberando toda la memoria utilizada por la instancia.

Se debe lograr que la implementación del TAD garantice que la función **pokerdeck_length()** sea de orden constante O(1). Además **el programa resultante no debe dejar *memory leaks ni lecturas/escrituras inválidas*.**

En el archivo **`pokerdeck.h`** se encuentran descripciones de las funciones enumeradas anteriormente y además se especifican las pre y post condiciones para cada una. Se deben verificar las pre y post usando **`assert()`** así como también deben 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. Una vez compilado el programa puede probarse ejecutando:

```
$ ./testdeck
```

Consideraciones:

- Solo se debe modificar el archivo **`pokerdeck.c`**
- Se incluyen un par de funciones **`static`** para manejo de nodos que pueden resultar útiles a la hora de completar el TAD. La función **`pokerdeck_push()`** (la cual ya viene resuelta) usa **`create_node()`** por lo que lo mejor es que las implementen y las aprovechen cuando sean de ayuda en otras funciones del TAD.
- 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 parcial.
- Los *memory leaks* bajan puntos
- Entregar código muy improlijo puede restar puntos
- Si **`pokerdeck_length()`** no es de orden constante *O(1)* baja muchísimos puntos
- Si **`pokerdeck_add()`** es de orden constante suma puntos pero no es obligatorio.
- Si **`pokerdeck_count()`** es de orden constante suma puntos pero no es obligatorio.
- Mientras más elaborada sea la invariante de representación más puntos se suma.
- **No modificar los .h puesto que solo se entregará pokerdeck.c**