FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico2

Lucas Franciulli

[7541/9515] Algoritmos y Programación II

Segundo cuatrimestre 2021

Nombre:	Franciulli, Lucas
Padrón:	107059
Email:	lfranciulli@fi.uba.ar

Índice

1. Introducción

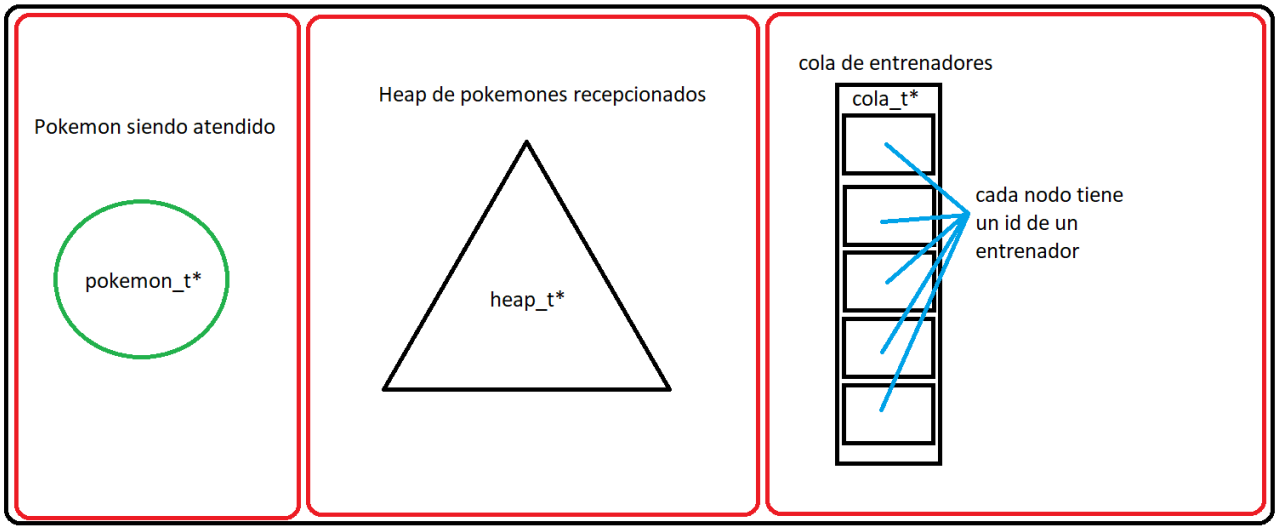
El objetivo de este informe es explicar la implementación de ciertas funcionas y la decisión de usar ciertos TDAs.

2. Structs

simulador_t:

```
34 ▼ struct _simulador_t{
35       cola_t* sala_de_espera_entrenadores;
36       heap_t* pokemones_recepcionados;
37       pokemon_t* pokemon_siendo_atendido;
38       unsigned pokemones_atendidos;
39
40       EstadisticasSimulacion* estadisticas;
41       lista_t* dificultades;
42       lista_t* funciones_dificultades;
43       unsigned cantidad_intentos;
44       hospital_t* hospital;
45
46       bool simulacion_finalizada;
47   };
```

El struct simuladot_t es el struct principal del archivo simulador.c. Este consiste en replicar de la mejor forma todo lo que pasa en el simulador.



En primer lugar, se encuentra la cola sala_de_espera_entrenadores en la cual se almacenan lo entrenadores antes de insertar sus pokemones en el heap. Use una cola ya que se debe mantener el orden de llegada de los entrenadores. Por ende, como una cola se basa en FIFO (First in, First out) me viene perfecto para este caso.

Luego tengo el heap que ya mencioné previamente. Decidí usar un heap minimal porque siempre tengo que elegir al pokemon de menor nivel entre todos los que hay en la sala de pokemones recepcionados. Para crear el heap decidí hacer todo un archivo heap.h y heap.c propio. En este se encuentran las funciones básicas de un heap: crear el heap, ver el nodo raíz, eliminar nodo raíz, agregar nodo y destruir heap. Además de todo esto, mi TDA heap funciona tanto como un heap minimal como un heap maximal. La diferencia entre estos es la función que se le pase al crear el heap la cual sirve para comparar entre dos nodos y decidir cual debe estar más cerca de la raíz. En el caso de mi simulador.c esto se da en la función pokemon_de_menor_nivel() en la cual compara dos pokemones y determina si el el primer void* que se pasa es menor al segundo o no.

Volviendo al struct simulador_t tengo un puntero al Pokémon que esta siendo atendido. La cantidad de pokemones ya atendidos en que se cuentan en la variable pokemones_atendidos. Después le sigue las estadísticas de simulador donde se almacenan los datos de la cantidad de pokemones atendidos, los que están en la sala de espera, etc. Luego se encuentran dos listas relacionadas a la dificultad. Una es dificultades, la cual almacena el nombre de la dificultad, el id y si esta en uso o no. Y la otra lista almacena las funciones relacionadas a cada dificultad. Mi decisión de que sean dos listas

separadas se debe a que en diferentes ocasiones del código tengo que recorrer una u otra debido a razones diferentes. Por ejemplo, si quiero saber que dificultad esta en uso recorro la primera lista y al momento de saber el id de la dificultad en uso simplemente utilizo este id como posición de la otra lista funciones_dificultades así accedo directamente a la posición donde se encuentran las funciones relacionadas a esta.

Por último, se encuentran la cantidad de intentos que le esta llevando a la enfermera adivinar el nivel del Pokémon que esta atendiendo actualmente. También se encuentra un puntero al hospital que almacena la lista de pokemones y entrenadores. Y finalmente hay un booleano el cual me indica si la simulación fue finalizada o no.

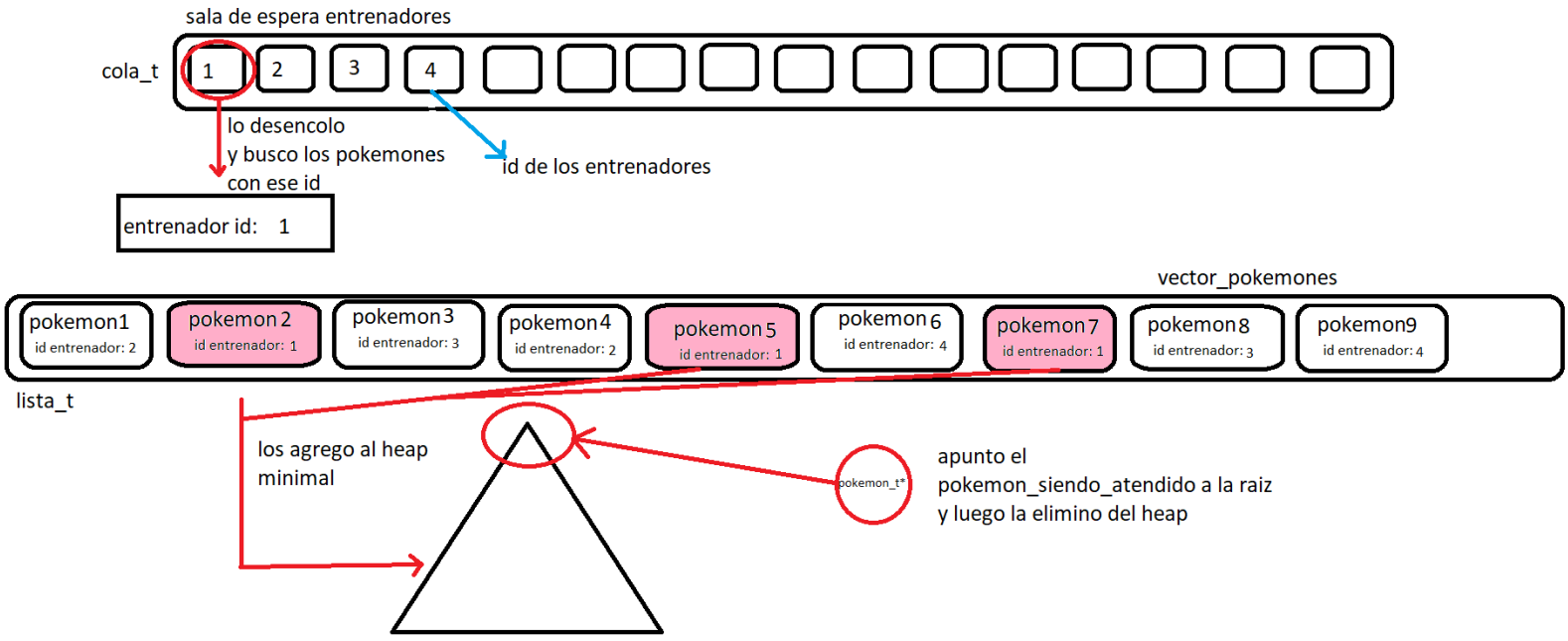
3. Funciones

simulador_simular_evento():

Esta función es la función de donde salen el resto de funciones. Al recibir un evento verifica cual de todos los eventos soportados es y llama a la función correspondiente. De no ser ningún evento soportado o de no haberse pasado un simulador valido, la función devuelve ErrorSimulacion. Si todo funciona bien devuelve Exitosimulacion.

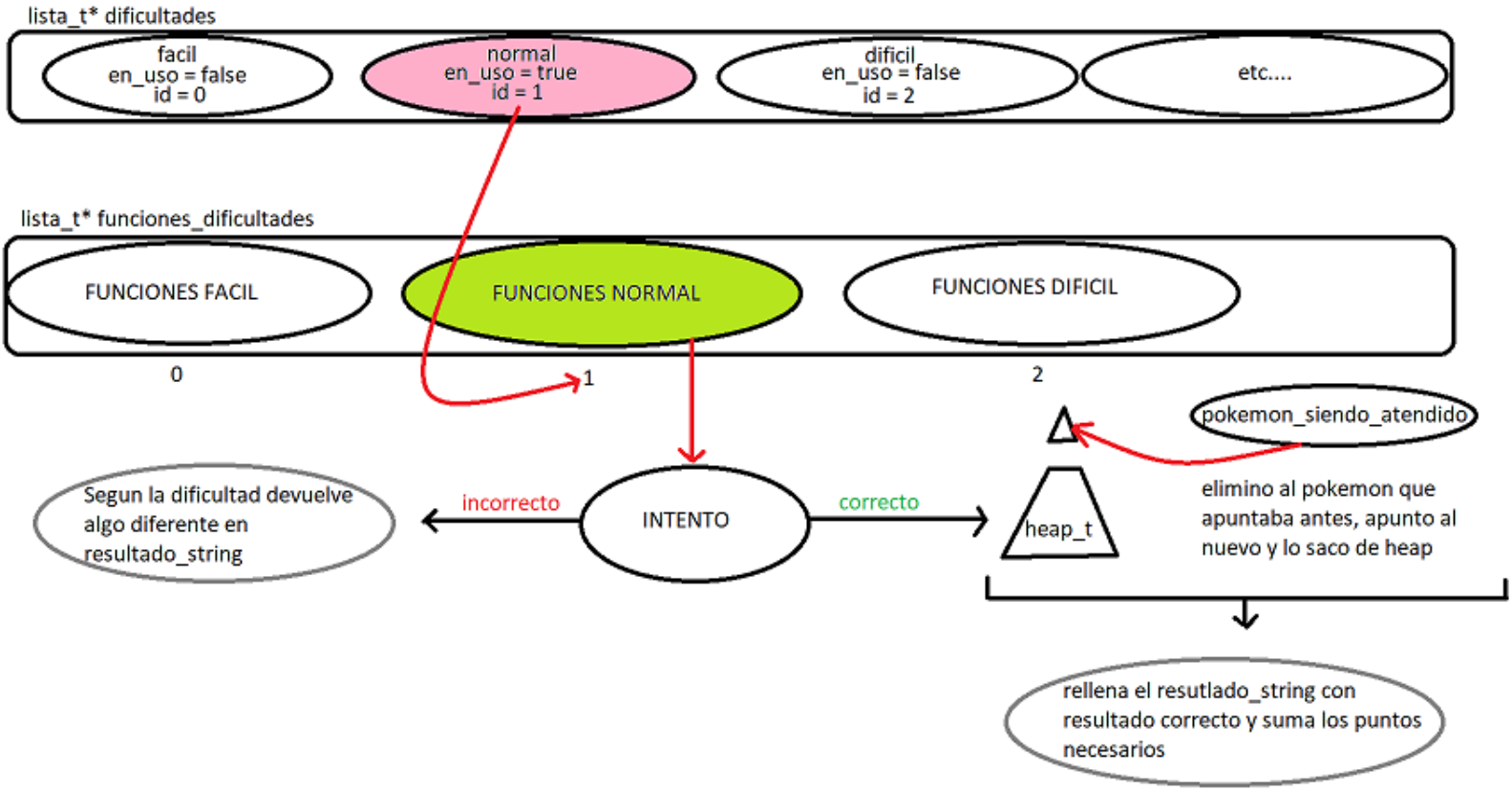
atender_proximo_entrenador ():

La función atender próximo entrenador primero desencola el primer elemento de la sala_de_espera_entrenadores para saber a cuál entrenador debe atender. Una vez tiene su id llama a la función agregar_pokemones_de_entrenador_a_heap() la cual se encarga de recorrer la lista vector_pokemones del hospital, donde se encuentran todos los pokemones de todos los entrenadores ordenados alfabéticamente. Acá lo que hacer es ir Pokémon por Pokémon verificando el entrenador de cada uno para ver si tienen el id del entrenador buscado. Si lo tienen se les hace una copia entera del Pokémon con la función copiar_pokemon() y se los inserta en el heap pokemones_recepcionados. Una vez insertados los pokemones de ese entrenador en el heap, si pokemon_siendo_atendido no esta apuntando a nada, se lo apunta a la raíz del heap y se saca ese pokemon del heap.



adivinar_nivel_pokemon():

La función adivinar_nivel_pokemon() recibe un intento de adivinar nivel. Primero de todo se encarga de ver que dificultad esta en uso. Para esto recorre la lista dificultades y ve cual es de todas. Una vez se sepa cual es se llama a la función adivinar_nivel_dificultad() siendo uno de los parámetros que se pasan el id de la dificultad que está en uso (el id es el mismo que muestra el main). En la función adivinar_nivel_dificultad() se encarga ir a la posición del id que paso la función anterior y aplicar las funciones de la dificultad que se encuentre en ese nodo. Si adivino el nivel del pokemon, lo libera, apunta pokemon_siendo_atendido a la raíz del heap y la elimina.



4. Main

El main.c empieza creando un hospital con un archivo llamado varios_entrenadores.hospital, luego crea un puntero al struct juego (el cual tiene un struct simulador y un booleano). Con todo esto llama a la función inicializar_juego() la cual crea el simulador y lo carga al struct juego y pone el booleano se_esta_jugando en true. Una vez terminado esto va a la función agregar_dos_dificultades() la cual crea dos dificultades nuevas asignándole sus nombres y apuntando a sus respectivas funciones.

Después entra en el bucle de pedir un input al usuario y dependiendo lo que este escriba llama a una función diferente. Los inputs aceptados son estos:

- (e) Mostrar estadísticas
- (p) Atender siguiente entrenador
- (i) Obtener información pokemon en tratamiento
- (a) Adivinar pokemon
- (d) Seleccionar dificultad
- (o) Ver dificultades
- (h) Mostrar comandos
- (q) Finalizar simulador

Mostrar estadísticas llama a la funcion simulador_simular_evento() con el evento obtener estadísticas y muestra por consola el valor de las estadísticas que devolvió.

Atender siguiente entrenador llama a la funcion simulador_simular_evento() con el evento AtenderProximoEntrenador e imprime por pantalla que se atendió el próximo entrenador.

Obtener información pokemon en tratamiento llama a la funcion simulador_simular_evento() con el evento ObtenerInformacionPokemonEnTratamiento e imprime por pantalla el nombre del entrenador y del pokemon siendo atendido actualmente.

Adivinar pokemon entra en un loop en el cual se le pregunta al usuario el nivel del pokemon que se esta atendiendo y por cada intento llama a la funcion simulador_simular_evento(), si adivino el nivel suma los puntos necesarios al simulador y de no ser así el loop continua. Dependiendo la dificultad imprime una pista diferente o resultado correcto si adivina correctamente.

Ver dificultades por cada una de las 5 dificultades llama a la funcion simulador_simular_evento() con el evento ObtenerInformacionDificultad. Con esto imprime el id de cada dificultad y al lado si esta en uso o no.

Seleccionar dificultad recibe un id de una de las dificultades que existen y la pone en uso llamando a la funcion simulador_simular_evento() con el evento SeleccionarDificultad.

Mostrar comandos muestra los comandos aceptados por el main.

Finalizar simulador llama a la funcion simulador_simular_evento() con el evento FinalizarSimulacion y cambie el valor de la variable se_esta_jugando cortando así el loop principal del main acabando su ejecución.