Trabajo Práctico 2 — Simulador

[7541/9515] Algoritmos y Programación II
Segundo cuatrimestre de 2021

Alumno:	Castillo, Carlos E.
Número de padrón:	108535
Email:	ccastillo@fi.uba.ar

Índice

1. Introducción

El trabajo practico presenta un hospital pokemon en el que los diferentes entrenadores pueden atender a sus pokemones. Inicialmente se provee la información de los entrenadores ingresantes y sus pokemones a traves de un archivo de texto. El objetivo es implementar la funcionalidad de atencion de los pokemones en el hospital a traves de un simulador que permita gestionar a los pacientes internados.

Dicho simulador toma control de la informacion del hospital y permite simular una serie de eventos a traves de diferentes comandos, que ayudan a las enfermeras del hospital a atender a cada pokemon adivinando su nivel.

Para la construcción del simulador del el hospital de pokemones se deben utilizar algunas de las diferentes estructuras de datos estudiadas durante el curso de la materia, identificando cual tda resulta mas conveniente para implementar cada funcionalidad del simulador.

2. Detalles de implementación

La implementación de esta estructura de datos fue escrita en el lenguaje de programación C, siguiendo la especificación del lenguaje detallada en el estándar C99. Los archivos principales se encuentran dentro del directorio **src** ubicada en la raíz del repositorio.

Además se incluye un archivo **pruebas.c** en el que se encuentran diferentes pruebas automatizadas para detectar errores en la ejecución del programa o en la asignación, manejo y liberación de memoria dinámica. Para esto se utiliza **gcc** como compilador y **valgrind** como herramienta de análisis de memoria.

El trabajo practico se encuentra dividio en dos componentes principales: la interfaz y el simulador. La implementacion de la interfaz abarca los archivos **main.c** y **juego.c**. Por otra parte, la implementacion del simulador se divide en los archivos **simulador.c**, en donde se encuentra la funcionalidad base del simulador, y los archivos auxiliares **aux_simulador_atencion_pokemon.c** y **aux_simulador_dificultades.c**, que almacenan funcionalidad comun utilizada en bastantes partes del simulador.

2.1. Simulador

El simulador provee tres funciones principales: `simulador_crear`, que se encarga de la creacion e inicialización de un nuevo simulador con su respectivo hospital, `simulador_simular_evento`, que evalua los diferentes comandos enviados al simulador y ejecuta el evento correspondiente a dicho comando, y finalmente `simulador_destruir` que se encarga de la destruccion del simulador y liberacion de la memoria ocupada por el mismo.

La estructura propuesta para el simulador es la siguiente:

```
struct simulador {
    hospital_t* hospital;
    EstadisticasSimulacion estadisticas;

    heap_t* recepcion;
    lista_iterador_t* sala_espera_entrenadores;
    lista_iterador_t* sala_espera_pokemones;
    PokemonEnRecepcion* pokemon_en_tratamiento;

    abb_t* dificultades;
    DatosDificultadConId dificultad_en_uso;
    unsigned intentos_actuales;

    bool en_curso;
};
```

El primer campo `hospital` hace referencia al hospital que se utiliza con el simulador. Este ultimo obtiene la informacion de los registros de ingresos de entrenadores y pokemones guardados previamente en el hospital antes de que estos sean cargados por turnos a la simulacion. El campo

`estadisticas` le permite al simulador llevar control sobre los el estado actual de la simulacion, almacenando datos como el numero total de entrenadores y pokemones, la cantidad de pokemones atendidos y en espera, entre otras estadisticas de la simulacion en curso.

Los campos `dificultad_en_uso`, `intentos_actuales` y `en_curso` permiten al simulador mantener una referencia a ciertos datos del estado actual de la simulacion que son de utilidad en algunos de los eventos registrados en el simulador.

2.2. Atención de Pokemones

2.2.1. Registro en Hospital

Inicialmente los entrenadores esperan pacientemente junto con sus pokemones en la sala de espera del hospital. Para almacenar esta informacion en el hospital se utilizan tres estructuras de datos diferentes: dos listas enlazadas y un árbol binario de búsqueda. Las listas enlazadas son de utilidad para guardar un registro de los entrenadores que van ingresando al simulador en el orden de llegada de los mismos al hospital, que es el mismo orden en el que los entrenadores son cargados al simulador. Al utilizar una lista enlazada este orden se respeta, ya que los elementos se almacenan en nodos dispuestos uno tras otro de manera consecutiva, en el orden en el que van siendo insertados. Precisamente por esta razón, en este caso la operación de inserción en la lista enlazada solamente se hace al final de la lista para cada nuevo elemento, en vez de hacer una inserción en posiciones específicas.

Además de un registro lineal de los entrenadores se necesita un registro lineal de los pokemones, de tal manera que sea facil relacionar un rango de pokemones con un mismo entrenador, en base a la cantidad de pokemones que registró cada entrenador al ingresar al hospital. Para esto se utiliza la segunda lista enlazada. Finalmente, el árbol binario de búsqueda se utiliza para facilitar la tarea de recorrer los pokemones en orden alfabético, utilizando un árbol binario que implemente un comparador basado en los nombres de los pokemones para que al realizar un recorrido "inorden" resulte en un recorrido en orden alfabético de todos los pokemones. Si bien esta decisión implica duplicar los datos de los pokemones almacenados, se decidieron priorizar las optimizaciones en cuanto a la complejidad temporal sobre la complejidad de memoria, esto haciendo referencia al alto costo temporal que tendría ordenar una lista enlazada de pokemones solo para recorrerlos en orden alfabético.

2.2.2. Ingreso de Entrenadores

Como fue mencionado anteriormente, el orden en el que "entran" los entrenadores en el hospital es el mismo orden en el que estos van a ser cargados, junto con sus pokemones, en el simulador. El simulador cuenta con un evento llamado "AtenderProximoEntrenador" el cual carga la información del siguiente entrenador no atendido cada vez que se invoca dicho evento, es decir, de cierta forma se va avanzando sobre el listado de entrenadores en orden de llegada, cada vez que se atiende un entrenador, se avanza al siguiente, y luego al siguiente, así hasta haber atendido a todos los entrenadores posibles. Como el registro de entrenadores en el hospital está implementado utilizando una lista enlazada que respeta este mismo orden, resulta conveniente utilizar un iterador de lista enlazada que permita replicar esta acción de avanzar al próximo entrenador de la lista cada vez que se termine de atender a un entrenador. Dicho iterador corresponde al campo `sala_espera_entrenadores` presente en el simulador.

Esta funcionalidad está implementada en la función `simulador_atender_proximo_entrenador` que toma el elemento actual del iterador de entrenadores, es decir, el próximo entrenador a atender, y agrega todos sus pokemones a la recepción del simulador. Para identificar cuáles son los pokemones del hospital que pertenecen a cada entrenador, al momento de registrar a cada entrenador se almacena como dato la cantidad de pokemones que trajo consigo cuando ingresó al hospital. De esta forma, para un entrenador con x pokemones, se pueden realizar x iteraciones sobre la segunda lista enlazada que almacena los pokemones del hospital en orden de ingreso, nuevamente utilizando un iterador de lista enlazada, en este caso almacenado en el campo `sala_espera_pokemones` . Al utilizar un iterador sobre la lista enlazada de pokemones, se evitan posibles problemas que

podrían ocurrir si se usaran índices para acceder a los pokemones por entrenador en una sola lista, ya que cada vez que se avanza el iterador, el elemento actual del mismo se deja en la posición que corresponde al inicio del rango de pokemones del siguiente entrenador, asegurando así que los pokemones de un entrenador "interfieran" con los pokemones de otro por un error de indexado. Además, esta implementación de iterador + contador de pokemones (por entrenador) permite reducir la complejidad del código y la complejidad temporal que se tendría de utilizarse otra estructura de datos como una doble lista enlazada (una lista enlazada de pokemones por cada entrenador en una lista enlazada de entrenadores). Para esto último se utilizan las funciones auxiliares `simulador_atender_pokemon_de_menor_nivel` y `aux_agregar_pokemones_de_entrenador_a_recepcion` que ayudan a modularizar un poco el proceso de guardado de los pokemones de cada entrenador.

2.2.3. Recepción de Pokemones

El simulador está diseñado para que cada vez que se atienda a un nuevo pokemon, siempre se elija aquel pokemon que tenga el menor nivel posible entre todos los ya cargados al simulador. Esto implica que cada vez que se atiendan nuevos entrenadores y estos pasen sus pokemones al simulador, dichos pokemones se deben almacenar en una estructura de datos que sea capaz de mantener el orden tras cada inserción, de tal forma que se garantice que en cuanto haya espacio disponible en el consultorio para atender a un pokemon, sea efectivamente el pokemon de menor nivel el que pase a ser atendido, sin verse afectado por otros factores que podría afectar el orden en ciertas estructuras de datos como el orden de llegada del pokemon.

Una vez establecida la necesidad de cumplir con este criterio, se hace evidente que la estructura de datos que encaja mejor con la funcionalidad requerida es el heap binario minimal. Esta estructura de datos es un árbol binario que mantiene en su raíz al menor de todos los elementos basados en el criterio de comparación deseado, en este caso el nivel del pokemon. De esta forma, cada vez que se desea seleccionar el próximo pokemon a ser atendido, solo se extrae la raíz del heap, que en el caso del simulador corresponde al campo `recepcion`, y directamente se asegura que dicha elemento extraído es el pokemon de menor nivel entre los pokemones disponibles en toda la recepción.

Para atender a un pokemon enlistado cuando hay espacio disponible en el consultorio, se utiliza la función auxiliar `simulador_atender_pokemon_de_menor_nivel`, que tal y como su nombre indica, una vez que se tiene espacio disponible para avanzar al pokemon de menor nivel para que este sea atendido, se encarga de destruir el pokemon que estaba actualmente en el consultorio (si es que había uno), extrae la raíz del heap que almacena todos los pokemones disponibles, y posiciona al pokemon de menor nivel como nuevo pokemon en tratamiento. En el caso del simulador, este pokemon es almacenado en el campo `pokemon_en_tratamiento`.

2.3. Manejo de Dificultades

El objetivo del simulador es permitir que el usuario adivine el nivel de un pokemon cuando este está siendo atendido. Esto es posible a través del evento "AdivinarNivelPokemonEnTratamiento". Este evento utiliza las funciones específicas de la dificultad que esta activa en el momento en el simulador para determinar si el intento de adivinar el nivel del pokemon hecho por el usuario es correcto o incorrecto. Para esto es necesario tener una forma de identificar cuál es la dificultad que se encuentra activa en el momento de realizar el intento y cuales son las funciones de esa dificultad para calcular los puntajes, generar pistas y verificar los resultados de cada intento.

El simulador cuenta con un registro de las diferentes dificultades agregadas al mismo, teniendo siempre como mínimo tres dificultades: "Facil", "Media" y "Dificil". Para almacenar estas dificultades en el campo `dificultades` del simulador, se hace uso de un árbol binario de búsqueda, principalmente por la eficiencia de esta estructura de datos para buscar cierto elemento en base al criterio de ordenamiento, en este caso, buscar dificultades en base al ID de las mismas. Sin embargo si bien sobre el papel esta estructura presenta esta ventaja, en la práctica no resulta ni más ni menos eficiente para la búsqueda que lo que hubiera resultado si se utilizara una lista enlazada, dado a que a como están construidas las funciones que ayudan a la creacion de una nueva

dificultad, `simulador_agregar_dificultad` junto con otros auxiliares como `crear_dificultad`, el ID de cada nueva dificultad agregada se auto-genera a partir de la cantidad de dificultades ya existentes, de tal manera que el ID de la nueva dificultad siempre va a ser mayor que los IDs existentes y por lo tanto el árbol binario de búsqueda se termina degenerando en una lista enlazada. Sin embargo se decidió conservar la implementación de árbol binario en caso de que se desee cambiar la auto-generación de los IDs en un futuro y también porque provee directamente la primitiva `abb_buscar`, para buscar una dificultad por ID cuando se desea cambiar la dificultad del simulador.

Una vez se cambia la dificultad del simulador, al momento de intentar adivinar el nivel del pokemon en tratamiento nuevamente, el evento tomará en consideración las funciones de verificación del intento correspondientes a la dificultad con el ID de la dificultad activa, almacenado en el simulador dentro del campo `dificultad_en_uso`. El simulador también lleva control sobre la cantidad de intentos requeridos hasta adivinar el nivel de un pokemon cada vez que se inicia dicho evento. Este datos es requerido por algunas dificultades para calcular el puntaje final.

2.4. Interfaz

La parte de la interfaz hace las veces de cliente para el simulador.

El campo `casillas` de cada hash es un vector de listas simplemente enlazadas y puramente recursiva, implementadas de la siguiente forma:

```
typedef struct casilla {
    char* clave;
    void* elemento;
    struct casilla* siguiente;
} casilla_t;
```