FACULTAD DE INGENIERIA

Universidad de Buenos Aires

Trabajo Práctico N°1 — HOSPITAL POKEMÓN

[7541/9515] Algoritmos y Programación II

Segundo cuatrimestre de 2021

Alumno:	BENITEZ, Nahuel Tomas
Número de padrón:	106841
Email:	ntbenitez@fi.uba.ar

1. Introducción

El TP busca a partir de uno o más archivos de texto en formato .csv que poseen información sobre cada entrenador(id y nombre) y sus pokemones(nombre y nivel), leer su información y almacenar la que fuera necesaria(nombre pokemon, nivel, cantidad pokemones y cantidad entrenadores).Se pide que el sistema pueda almacenar la información de los archivos. Debe ser posible abrir uno o más de los archivos con el formato pedido y luego buscar en el hospital la información básica.

Para esto, se fue leyendo línea a línea el/los archivo/s y guardando todo su contenido en un buffer que vive en el heap, con la posibilidad de reservar más memoria en caso de ser necesario, e inmediatamente se enviaba a una función split() que separaba la información contenida entre los ";", en distintos substrings apuntados por un único puntero. Luego se reservaba la memoria necesaria en el heap para guardar cada substring requerido en el struct hospital, y así repitiendo hasta que se termine de leer todo el archivo. Una vez cargado se liberaba el buffer y el puntero a punteros que devolvía la función split().

También se llevaron a cabo otras funciones puntuales, para finalizar liberando toda la memoria que se había reservado en el programa para el arreglo hospital.

2. Detalles de la implementación

1. Detalles de algunas funciones:

char split(char* string, char separador);** La función split recibe un string y un separador, y devuelve un vector dinámico de substrings, siendo cada substring el contenido que se encontraba entre separadores, o entre un separador y el final del string, en el string recibido.

hospital_t* hospital_crear();

Reserva un bloque de memoria en el heap de tantos bytes como tenga hospital_t, para hospital, lo inicializa con ceros, y devuelve un puntero al inicio del bloque de memoria reservado. En caso de error devuelve NULL. *Diagrama 1.*

void reservar_memoria_para_pokemon(hospital_t* hospital, size_t pokemones_a_agregar);

Si los pokemones a asignar son los primeros del vector, crea un bloque de memoria en el heap de tamaño igual al tamaño en bytes de pokemon_t multiplicado por pokemones_a_agregar + 1, y lo asigna al puntero *vector_pokemones. Si ya se hubiesen asignado otros pokemones, usaría un realloc() para agrandar el bloque de memoria de pokemon_t* con el tamaño necesario para ingresar todos los pokemones_a_agregar. Se asigna la memoria reservada a un pokemon_t* auxiliar primero, porque si llegara a fallar, auxiliar quedaría apuntando a NULL, y así no perdería el puntero a hospital->vector_pokemones. Como, en cambio, sí pasaría si lo asignase directo a calloc() a hospital->vector_pokemones. *Diagrama 2.*

void reservar_memoria_nombre_pokemon(hospital_t* hospital, size_t longitud_string, size_t posicion);

Reserva un bloque de memoria con calloc() para cada nombre de cada pokemon en el hospital. Recibe la longitud del string del nombre del pokemon actual, y reserva longitud_string + 1 bytes en el heap, un byte para cada char del nombre, y un byte para el '\0' del fin del string. Lo asigna a la posición actual en el vector_pokemones del pokemon en cuestión. *Diagrama 3.*

void agregar_a_hospital(hospital_t* hospital, char linea_leida, size_t pokemones_a_agregar);**

Agrega en hospital los datos devueltos por split() a char** linea_leida. Sabiendo a partir de la tercera posicion de linea_leida (linea_leida[2]), cada dos posiciones de linea_leida se encuentra primero el nombre del pokemon (linea_leida[2]) y luego su nivel (linea_leida[3]) se agregan según correspondan a hospital->vector_pokemones[k].nombre y hospital->vector_pokemones[k].nivel. } Uso la función atoi() que me permite obtener un entero a través de un string para conseguir el nivel. Y uso strcpy para copiar el valor de linea_leida[] donde haya un nombre, uso esta función ya que si le asignase (hospital->vector_pokemones[k].nombre = linea_leida[];) , estaría enviando una dirección de memoria que luego voy a liberar para leer otra línea del archivo.

void liberar_linea_leida(char linea_leida);**

Libera la memoria que ocupa cada substrings de linea_leida[k], y luego libera la memoria que ocupa linea_leida. Si no se liberara primero cada substring y se liberara directamente linea_leida, se liberaría el bloque grande y se perderían los punteros a los bloques de memoria de los substrings, que no se habrán liberado. *Diagrama 4.*

bool hospital_leer_archivo(hospital_t* hospital, const char* nombre_archivo);

Abre el archivo, lo lee línea a línea con un fgets() guardando la línea leída en un buffer dinámico, en caso de que la línea no entre en el buffer, este se agranda con un realloc() para que entre bien. Luego manda el buffer y el separador usado en el archivo a la función split(buffer, ";"). Una vez terminado el proceso porque se llegó al final del archivo, y el fgets() devuelve NULL al tratar de leer, se libera el

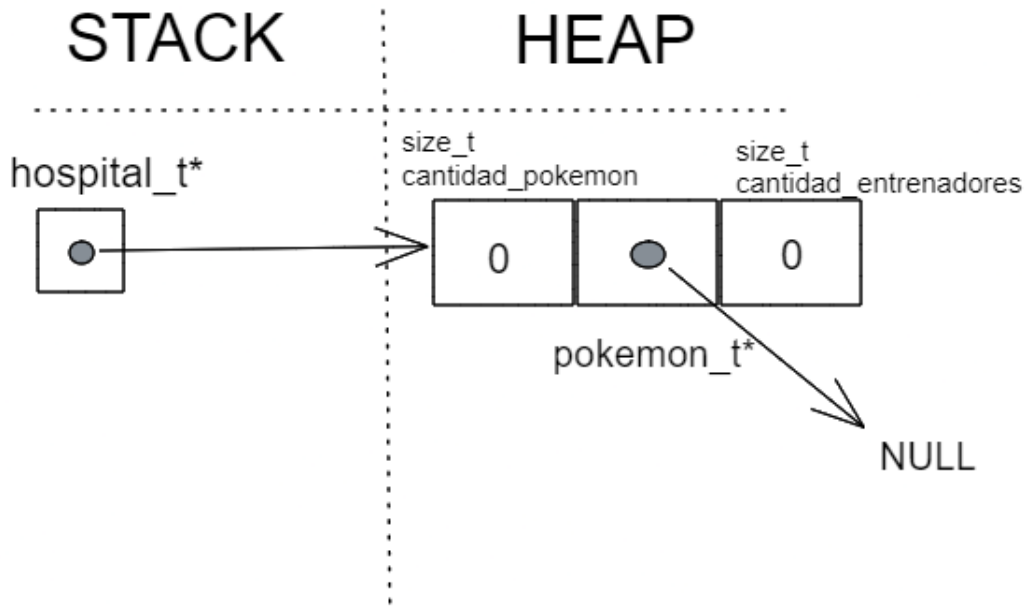
buffer y se cierra el archivo. Devuelve true si se pudo hacer lo anterior sin inconvenientes. Caso contrario devuelve false.

void hospital_destruir(hospital_t* hospital);

Libera el hospital y toda la memoria utilizada por el mismo. Libera primero el nombre de cada pokemon, luego libera el vector_pokemones y por último el hospital.

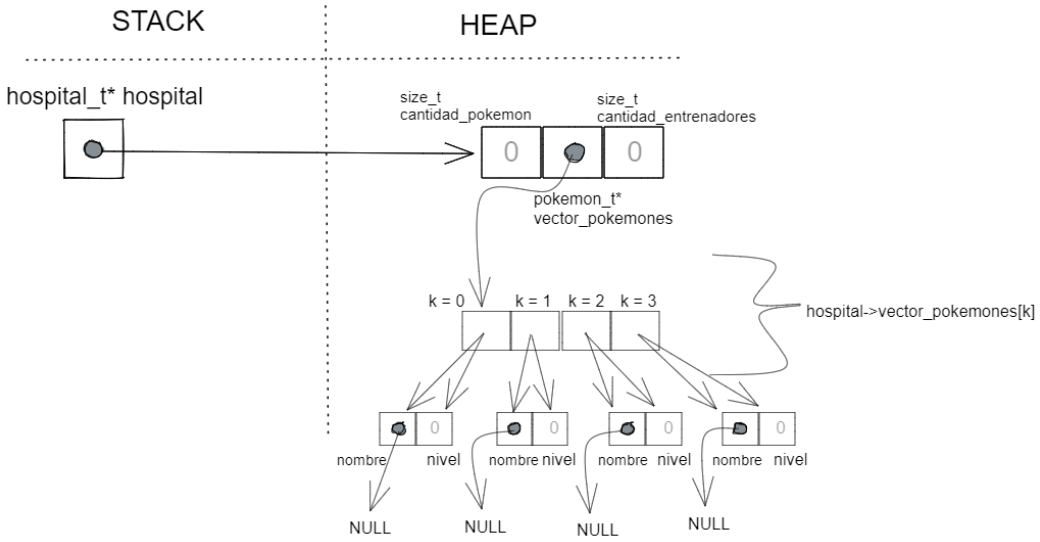
3. Diagramas

1. Diagrama 1



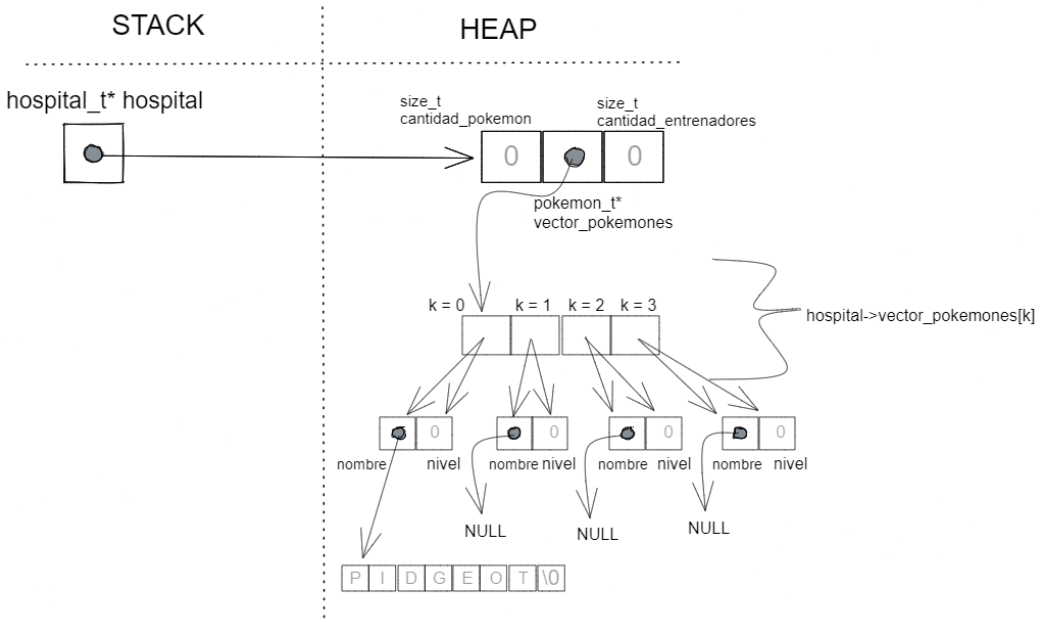
Ejemplo de creación de un puntero hospital_t*, creada en el stack y apunta a una dirección de memoria en el heap, esa memoria fue reservada por calloc(), que también inicializó las variables en 0 y el puntero en NULL.

2. Diagrama 2



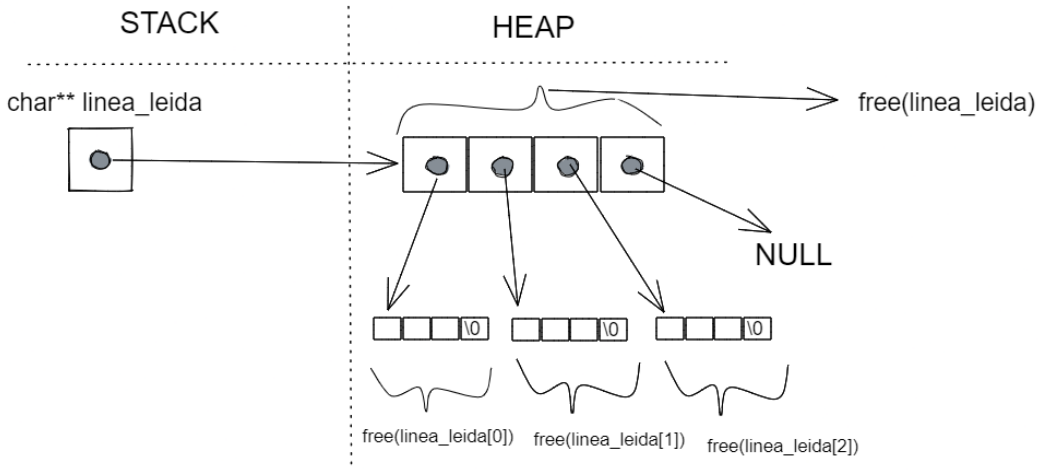
Ejemplo de creación de un bloque de memoria, de 4 pokemon_t mediante calloc().

3. Diagrama 3



Ejemplo de creación de un char* de 8 bytes, que guarda a un nombre de pokemon.

4. Diagrama 4



Ejemplo de como se libera la memoria reservada para linea_leida. Primero se deben liberar cada substring para luego liberar toda la memoria.