FACULTAD DE INGENIERIA

Universidad de Buenos Aires

Trabajo Práctico 2 — Simulador Pokemon

[7541/9515] Algoritmos y Programación II

Segundo cuatrimestre de 2021

Alumno:	**SANCHEZ FERNANDEZ DE LA VEGA, Manuel**
Número de padrón:	107951
Email:	msanchezf@fi.uba.ar

1. Introducción

La idea general del TP era realizar un simulador haciendo uso de los pasados trabajos prácticos. El TP2 se basaba en el hospital del TP1. Sin embargo, había que hacer cambios a la implementación debido a la restricción de no usar vectores, solo TDAs. En consecuencia, también tuve que usar los TDAs que desarrolle durante el año teniendo en cuenta cual es el mas optimo para usar en cada caso.

Luego de desarrollar el simulador, tuve que implementar al mismo en un pequeño juego utilizando la linea de comandos.

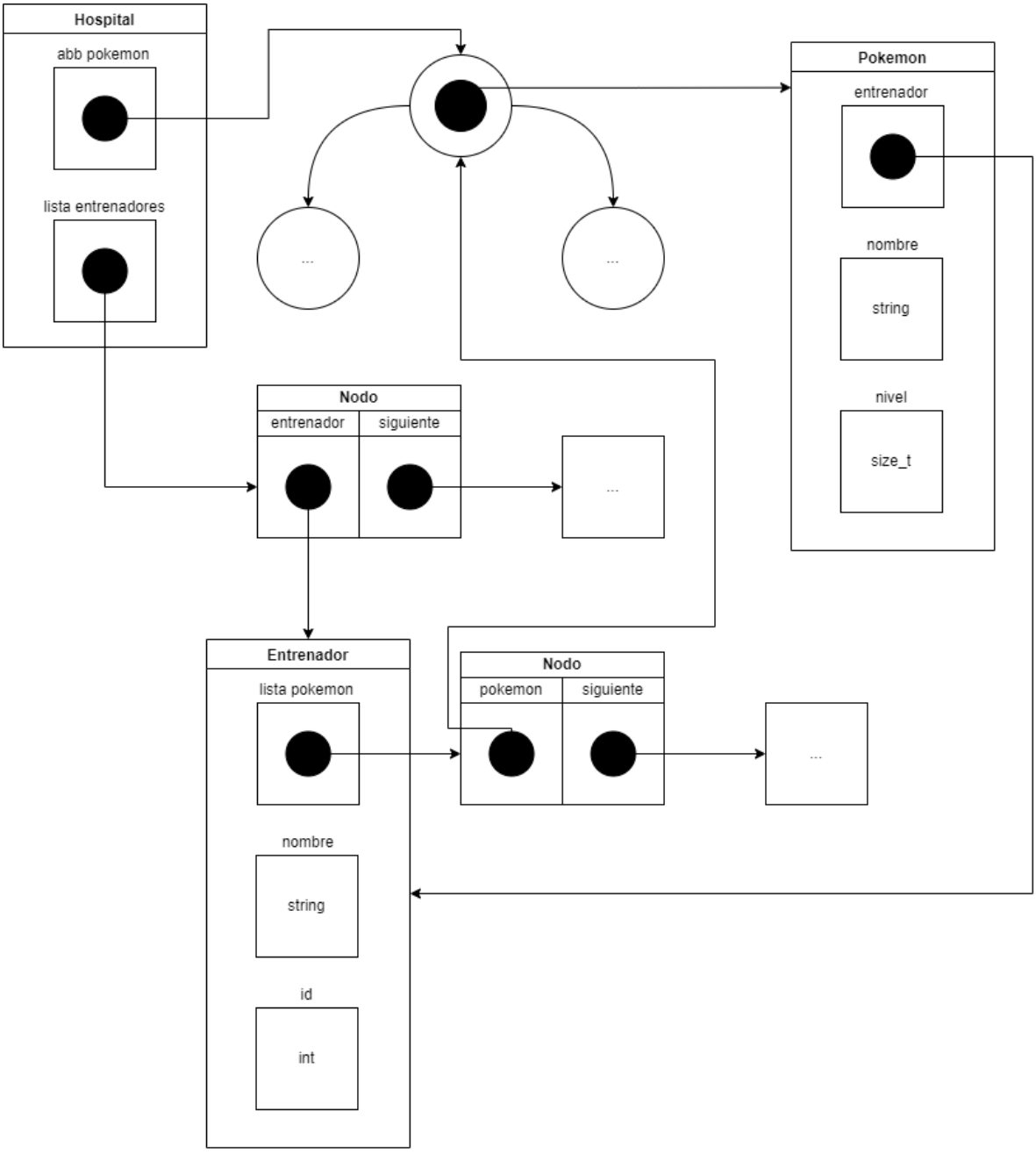
2. Detalles de implementación

Antes de implementar el TP2, tuve que hacer cambios a las estructuras del hospital del TP1. Para empezar, reemplace el vector de pokemon por un árbol binario. Esto se debe a que me viene muy bien a la hora de implementar la función de `lista_con_cada_elemento()` ya que esta recorre los pokemon y aplica una función pero en orden alfabético. En consecuencia, me conviene usar un ABB e insertar los pokemon según su nombre, operación cuyo orden de complejidad es O(log(n)) (suponiendo que el árbol esta balanceado, lo cual no siempre es cierto). Consecuentemente, puedo recorrer el ABB inorden para obtener los pokemon en orden alfabetico, sin necesidad de ningun tipo de ordenamiento extra.

Luego, para el vector de entrenadores, lo reemplace directamente con una lista. Esto es así ya que desde el punto de vista del hospital, los entrenadores no tienen porque estar ubicados en una cola (como lo van a hacer en simulador). Además, hace que algunas operaciones, como el copiado de la lista, sean mas simples que usar una cola. Esto me interesa ya que no es correcto modificar las estructuras de hospital desde simulador, pero puedo hacer copias en el simulador y modificarlas.

Asimismo, cambie algunos campos en las estructuras de pokemon y entrenador:

- A los pokemon, les agregue un puntero a su entrenador. Es decir, ahora puedo saber cual es el entrenador de cada pokemon teniendo únicamente la estructura del pokemon. Este campo reemplaza al campo de id_entrenador que había agregado en mi anterior implementación.
- Por el otro lado, a los entrenadores les agregue una lista de sus pokemon. Nuevamente, desde el punto de vista del entrenador, no tiene sentido tener un heap minimal de pokemon (de esto se encarga el simulador). Un entrenador quiere poder acceder a cualquiera de sus pokemon en cualquier momento, y no solo al de menor nivel. Este campo tiene como fin optimizar la búsqueda de todos los pokemon de cada entrenador. A la hora de atender un entrenador en el simulador, no es necesario iterar por todos los pokemon del hospital buscando cuales le pertenecen, sino que, dado un entrenador, podemos saber quienes son sus pokemon con un orden de complejidad muy reducido.



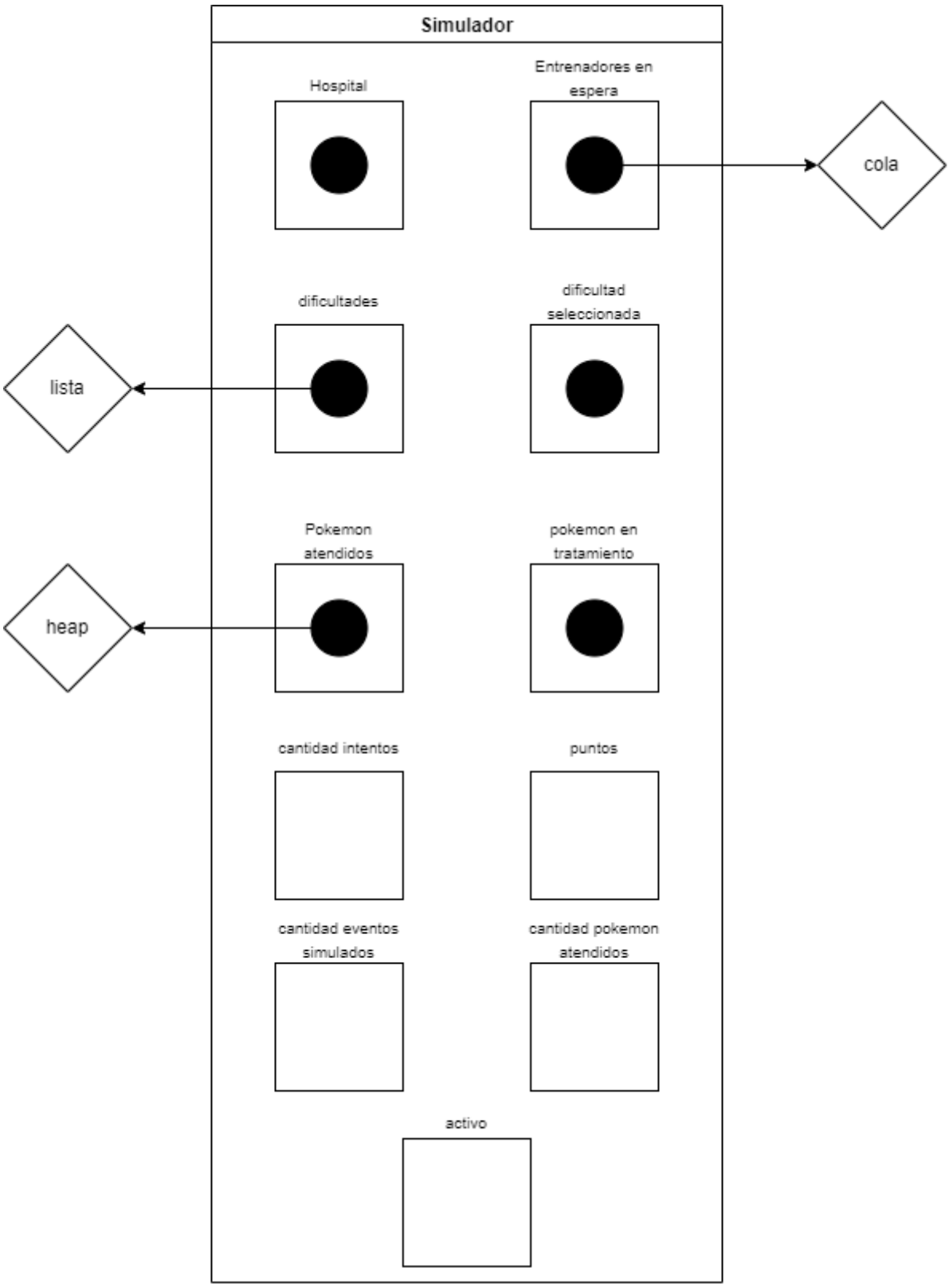
En la implementación del simulador, decidí agregar diversos campos. Muchos de ellos guardan estadísticas del simulador, como por ejemplo la cantidad de intentos a la hora de adivinar un nivel, los puntos, la cantidad de pokemon atendidos y la cantidad de eventos simulados.

No obstante, también agregue una cola de entrenadores a atender, en la cual los entrenadores esperan a su turno para entregar sus pokemon. Esta estructura es la indicada para esta situación ya que no me interesan todos los entrenadores, sino que solamente el que esta próximo a ser atendido.

Además, implemente un heap minimal de pokemon, ordenado según el nivel de cada uno. Esto se debe a que la complejidad de obtener el pokemon de nivel mas bajo es O(log(n)), lo cual es muy útil para el simulador ya que siempre se debe tratar al pokemon de nivel mas bajo. Además, agregue un puntero al pokemon siendo tratado actualmente.

Finalmente, al momento de guardar las distintas dificultades, utilice una lista. Esto se debe a que la lista me permite acceder a todos sus elementos, a diferencia de una pila o cola (donde debo desapilar o desencolar para ver los del medio). Además, el orden de las dificultades no es de importancia, lo único que importa es el id de cada una, lo cual representa la posición en la lista. En un principio, había pensado usar un hash, cuya clave para cada elemento sea el nombre de la dificultad. Sin embargo, me encontré con el problema de que para obtener la dificultad, me pasan por parámetro el id de la misma en vez del nombre. El id básicamente representa el índice en la lista donde se lo guardo, por lo que aunque el hash sea mas eficaz en términos de búsqueda, no lo puedo usar en esta ocasión ya que no dispongo de una clave para buscar elementos. Al igual que con el pokemon en tratamiento, tengo un puntero que indica la dificultad seleccionada. Una alternativa a esto es tener el id de la dificultad seleccionada, sin embargo, cada vez que quiero atender un pokemon, debería recorrer la lista buscando este índice por lo cual no es del todo eficiente.

En resumen, opte por sacrificar tamaño en la memoria por velocidad. Tranquilamente se puede hacer todo el TP2 usando índices en la estructura de simulador, que indican una posición en las estructuras del hospital. Sin embargo, trabajar de esta forma es muy poco eficiente en cuanto al tiempo de ejecución, ya que para acceder a cada elemento, no tengo otra que iterar por toda la estructura hasta encontrar dicho elemento. Al agregar diversas estructuras como el heap de pokemon a ser tratados o la cola de entrenadores, los tiempos de ejecución se reducen mucho.



(Los rombos representan TDAs para hacer mas compacto el diagrama)

Los TDAs usados en mi implementación fueron:

- ABB
- Cola
- Heap
- Lista

El heap es el único que tuve que desarrollar ya que el resto son los que desarrolle en este curso. Sin embargo, ya tenia casi toda la implementación del heap hecha gracias a RPL, lo único que tuve que cambiar fue el hecho de que sea minimal y sirva con `void*`.

Los siguientes son detalles de implementación que me gustarían destacar:

1. `puntero_y_retorno_t`

   Cree esta estructura con el fin de poder reusar la función `lista_con_cada_elemento()`. Debido a que esta solo acepta un parámetro auxiliar y devuelve la cantidad de elementos iterados, esta es la única forma (aparte de modificar la firma de la función) para pasar un parámetro auxiliar y obtener el retorno de la función. Me interesa obtener el retorno de `lista_con_cada_elemento()` ya que esta devuelve la cantidad de llamados a función. Es decir, si la función llegara a fallar en el ultimo elemento, no tendría forma de saberlo. Por esta razón, es importante saber que devuelve la función pasada por parámetro.

2. Tamaño inicial del heap

   Opte por que el tamaño inicial del heap del simulador sea 6, ya que, al menos en la versión oficial de los juegos de pokemon, el máximo numero de pokemon que puede tener un entrenador en su mochila es 6. Entonces, el simulador esta preparado inicialmente para almacenar los 6 pokemon de 1 solo entrenador, si llegaran a ir mas entrenadores, el heap puede duplicar su tamaño.

3. Función de comparación del heap

   El heap requiere de una función de comparación para ubicar sus elementos en sus posiciones correspondientes. Es por eso que, al querer un heap minimal ordenado según los niveles de los pokemon, cree una función que compara los niveles de cada uno de los pokemon. Esta función es `comparador_nivel_pokemon()` en simulador.c

4. Suposición sobre entrenadores duplicados

   En mi implementación, los entrenadores duplicados se tratan como entrenadores diferentes. Desde mi punto de vista, son como distintas "visitas" al hospital, por lo que decidí tratarlo 2 entrenadores distintos. Además, no tiene mucho sentido agregar todos los pokemon de un entrenador a la vez ya que como había mencionado previamente, un entrenador solo puede tener 6 pokemon como máximo.

5. Suposición sobre id de dificultad

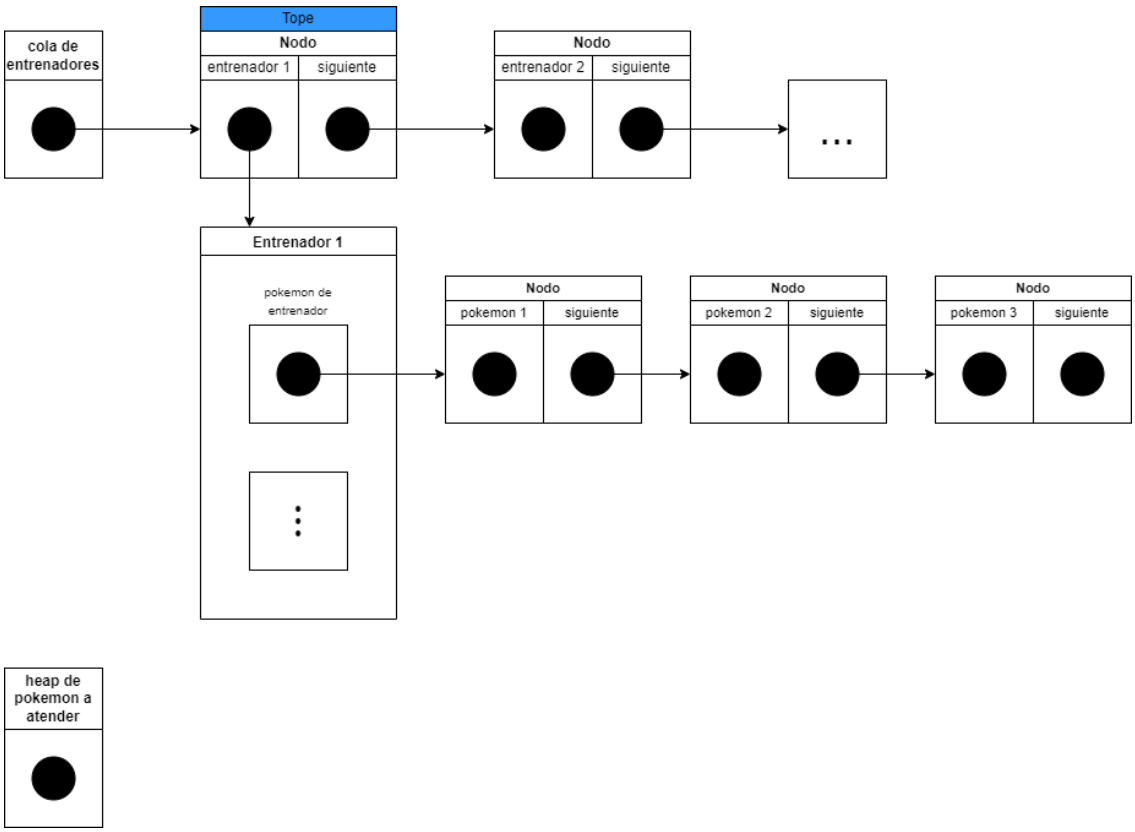
   Debido a que la dificultad se elige por id y no por nombre, cada vez que agrego una dificultad a la lista de dificultades, su posicion representa su id. En consecuencia, cada vez que quiero obtener una dificultad dado un id, simplemente accedo a la posicion del id de la lista

6. `leer_numero()`

   Esta función la utilice en el main, y su objetivo es leer el primer numero ingresado por el usuario. Esto significa que si el usuario ingresa "a 12 b 3", la función devuelve 12 como resultado. La utilizo a la hora de preguntarle un id o un nivel para adivinar al usuario. Su funcionamiento se basa en hacer un split al texto ingresado por el usuario, y buscar cual de esos substrings es un valor numérico. Si no encuentra ninguno, devuelve -1.

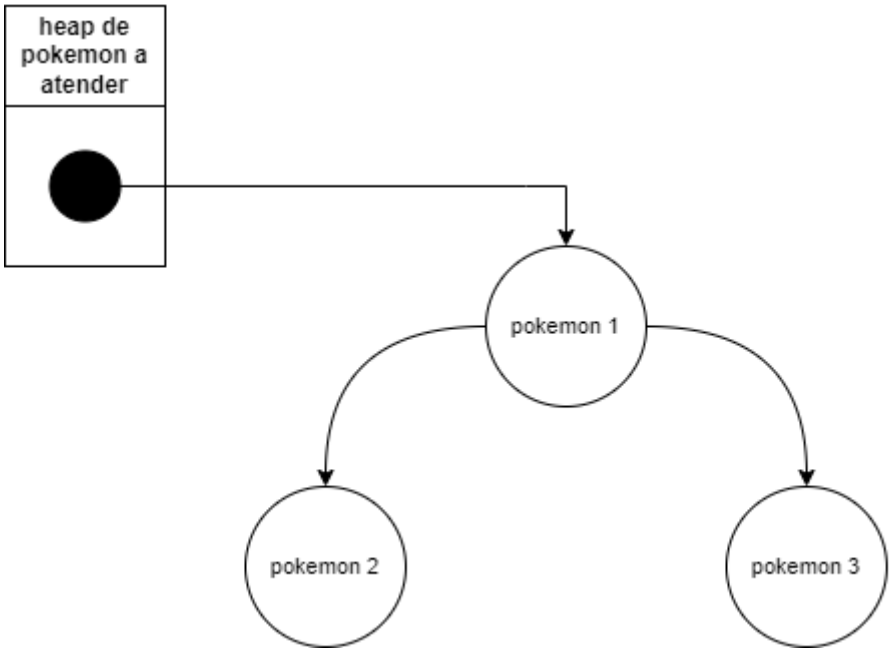
7. Explicación de las estructuras durante la simulación

   Considerando la siguientes estructuras del simulador

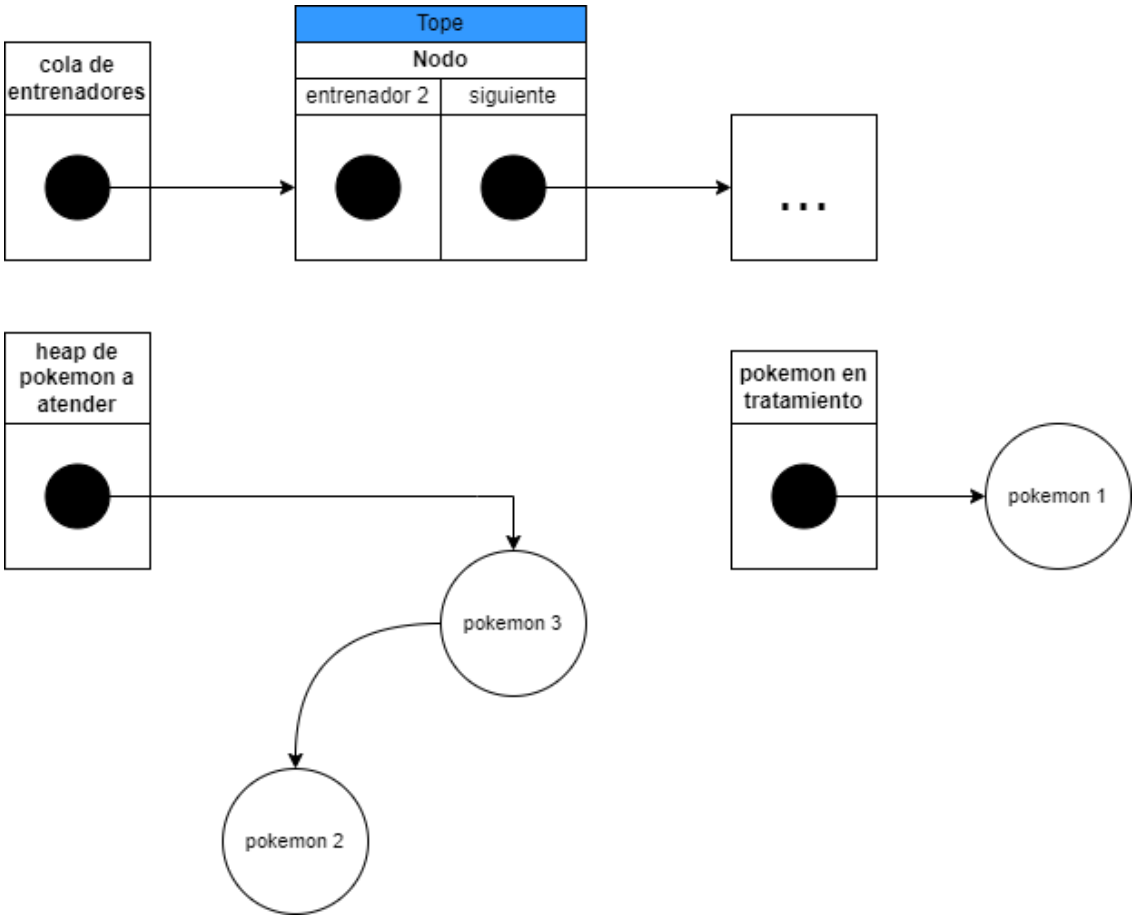


Al tratar al siguiente entrenador, simplemente desencolo el tope y agrego sus pokemon al heap, resultando en algo así:

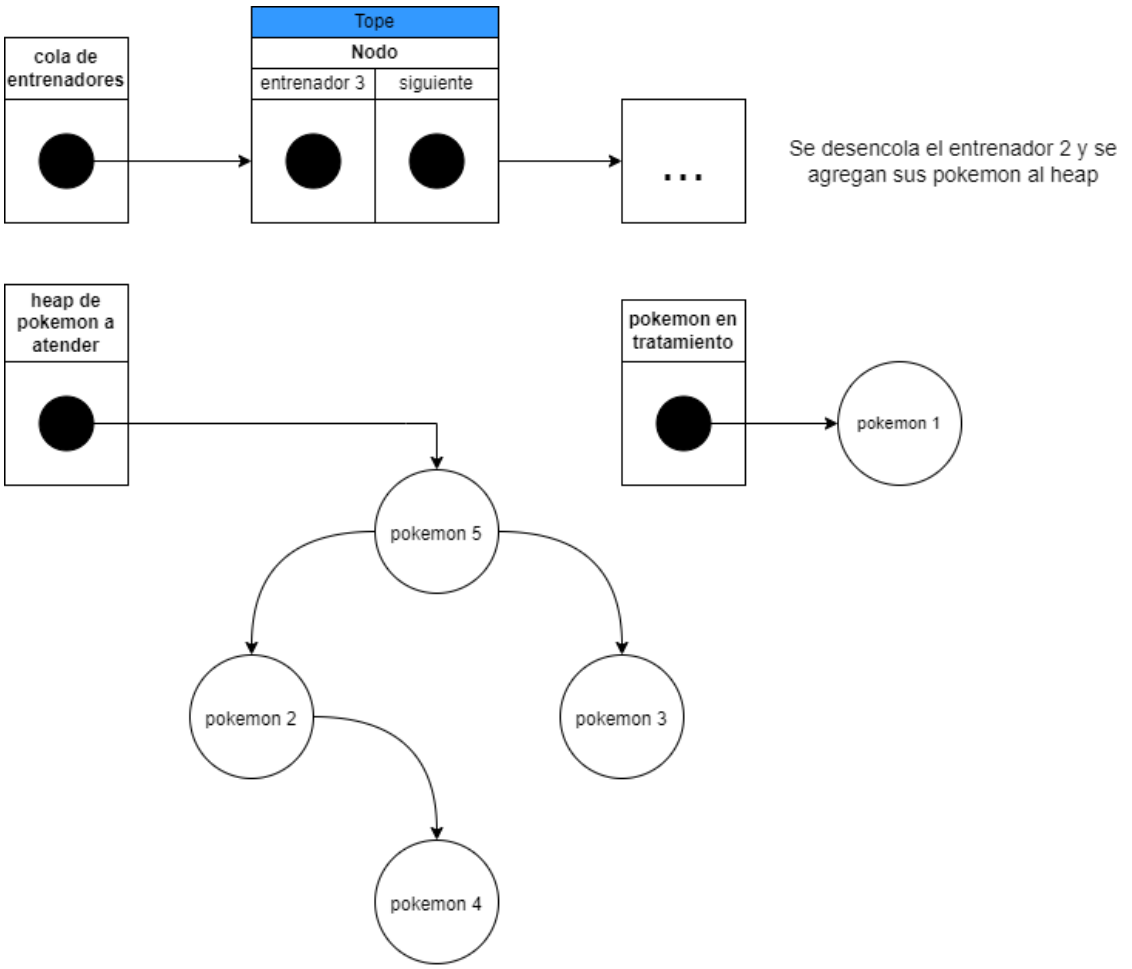
El entrenador 1 fue desencolado y sus pokemon fueron insertados en el heap



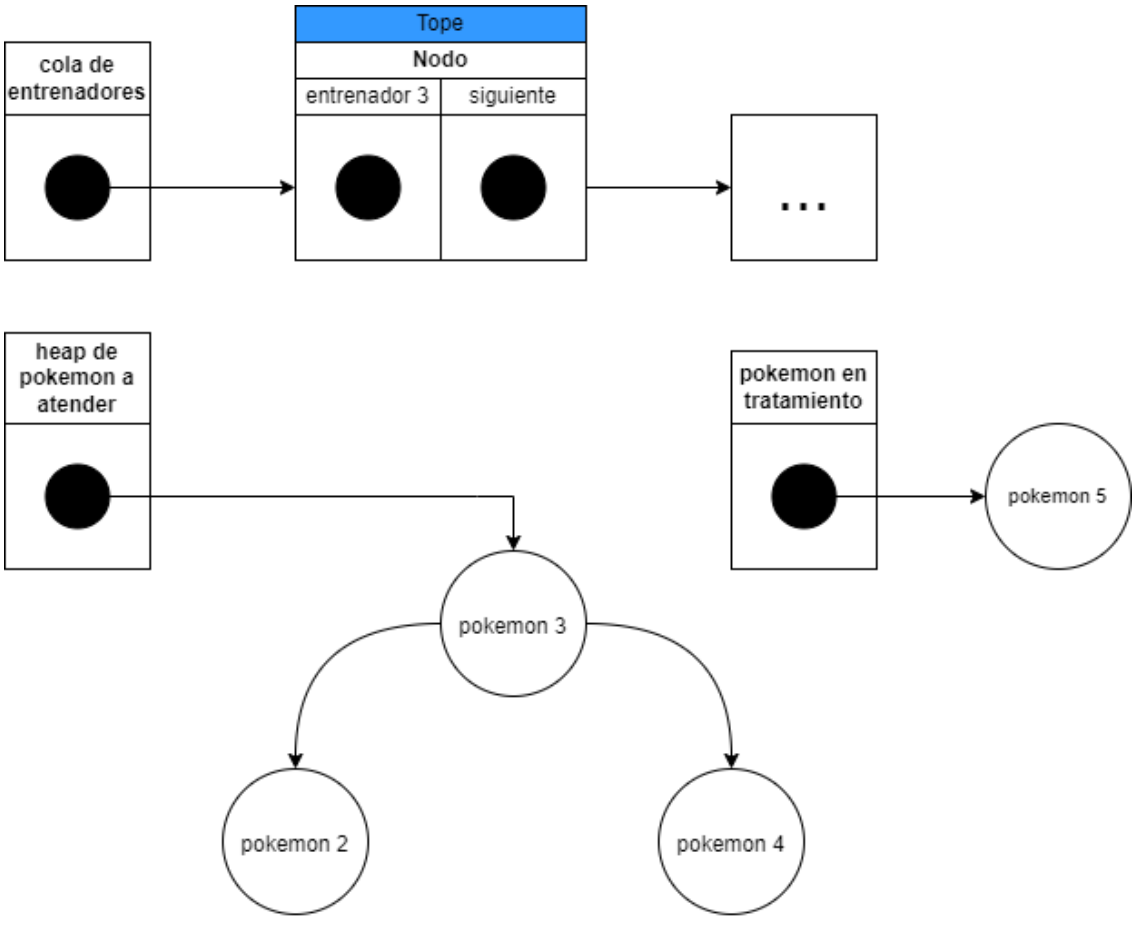
En este caso, el pokemon 1 es el menor de los 3 pokemon que tenia el entrenador. Como no hay ningun pokemon siendo tratado de momento, extraigo el pokemon de nivel mas chico del heap (el pokemon 1) y lo guardo como pokemon en tratamiento.



Ahora yo puedo continuar repitiendo el proceso y agregando pokemon de otros entrenadores.



Finalmente, una vez tratado el pokemon en tratamiento, se extrae el pokemon de nivel mas bajo del heap (en este caso es el pokemon 5) el cual pasa a ser el siguiente pokemon en tratamiento. Nuevamente, el pokemon 3 al ser menor que el resto (en este caso) se posiciona como el proximo a ser tratado.



3. Como correr el programa

Si se quiere correr el juego, hay que tener los siguientes archivos (tanto .c como .h) en una carpeta `./src`

- ABB
- Cola
- Heap
- Hospital y hospital interno
- Lista
- Split
- Simulador

Además, se debe contar con la carpeta de ejemplos con los archivos de entrenadores.

Luego, el `main.c` debe estar en el directorio principal junto con el makefile provisto. Para correr el main, hay que ejecutar el siguiente comando con la terminal: `make main`.

Por el otro lado, si se quiere correr las pruebas del simulador, se debe contar con el archivo `pruebas_sim.c` en el directorio principal y ejecutar `make valgrind-pruebas-sim`.

Finalmente, para correr las pruebas del hospital, se debe contar con el archivo `pruebas.c` en el directorio principal y ejecutar `make valgrind-pruebas`.

(Se necesita el archivo `pa2mm.h` para correr las pruebas)