TRABAJO PRÁCTICO 2

BERNARDOTTI, Tomás - Padrón: 105696
RUEDA, Nazarena - Padrón: 106280

Corrector asignado: Adeodato Simó

El objetivo que nos propusimos para este trabajo fue poder modularizar cada aspecto del programa, haciéndolo más prolijo, conciso, mantenible, y escalable en el tiempo.
Dividiremos este informe en cinco partes:

Primeros pasos: creación de la clínica

Decidimos crear un TDA ‘’Clínica’’ para poder almacenar todos los TDAs necesarios a la hora de llevar a cabo el TP. Nos resulta más prolijo hacerlo de esta forma porque en vez de tener varios TDAs creándose y destruyéndose en distintas partes del trabajo, con una función clínica_crear diseñamos todo lo que utilizaremos después. La clínica tiene como atributos todas los tipos abstractos de datos necesarios (y explicadas más adelante en el informe). Además, creamos dos *structs*: Paciente y Doctor. Estos nos sirven para guardar los diferentes atributos que correspondan a cada uno de una manera más “elegante”.

Todas las funciones de *funciones_tp2* son las primitivas de este TDA Clínica. Entre ellas se encuentran todos los algoritmos necesarios para llevar a cabo el funcionamiento de la clínica. Estas primitivas son utilizadas en *zyxcba.c*, en donde no se conoce de su funcionamiento interno (solo se encarga de cargar los CSVs y procesar los comandos recibidos por entrada estándar), si no únicamente de su utilidad.

Lectura de CSV y primer almacenamiento de datos recibidos

Decidimos almacenar a los doctores en un TDA ABB cuyas claves son el nombre de los doctores y el valor asociado su *struct* (en el que se encuentra toda la información relevante de cada uno). Decidimos utilizar este TDA porque así cumple con las complejidades pedidas para el Informe de Doctores.

Además, decidimos utilizar un TDA Hash para almacenar a los pacientes inscriptos en la clínica. Las claves son los nombres de los pacientes y los valores asociados sus *structs*. Elegimos hacer uso de este TDA por su baja complejidad temporal.

Al leer ambos CSVs, se crean los doctores y pacientes correspondientes y se almacenan en dos listas distintas. Al iterar estas listas, almacenamos cada elemento donde corresponda (o en el ABB de doctores o en el HASH de pacientes). Además, inicializamos los demás TDAs que utilizaremos a lo largo del trabajo.

Pedir Turno

Para este comando era necesario distinguir entre pacientes urgentes y regulares. Pedir un turno debe funcionar en *O(1)* en los casos urgentes, y puede relajarse para funcionar en *O(log n)* en casos regulares. Para lograr esto, pensamos qué tipos abstractos de datos podíamos usar para almacenar a los pacientes

(decidimos que era mejor diferenciar entre los pacientes regulares y los pacientes urgentes para que pudiéramos enfocarnos en las complejidades pedidas).

Creímos necesario el uso del TDA Hash para almacenar a los pacientes que pidieron un turno. Por un lado, tenemos un Hash cuyas claves son especialidades y cuyos valores asociados son *colas*; por el otro lado, el segundo Hash también tiene especialidades como clave pero como valor asociado tiene *heaps*. El Hash de las colas fue utilizado para almacenar a los pacientes urgentes una vez que pidieron turno en la especialidad especificada; el Hash de los heaps, para almacenar a los pacientes regulares.

Una cola encola y desencola en *O(1)*, lo cual es útil para que podemos almacenar en O(1) a un paciente urgente. Como el Hash también guarda elementos en O(1), de esta forma sigue manteniéndose la complejidad al pedir turno para pacientes urgentes. De la misma forma, pensamos que un heap era conveniente para almacenar a los pacientes regulares (dentro de la respectiva especialidad) porque el Heap encola y desencola en *O(log n)* -siendo *n* la cantidad de pacientes encolados en la especialidad dada-, por lo que sigue respetándose la complejidad al encolar a los pacientes que corresponden. Además, un heap, al ser una cola de prioridad, permite asignarle cierto orden a los elementos dentro de él. Para ello, (como pide la consigna), decidimos asignarle más importancia a los pacientes regulares que están inscriptos en la clínica desde hace más tiempo. Es decir, esa fue la prioridad elegida para llevar a cabo en la distribución de los elementos dentro del heap (el primero en desencolar es el más prioritario).

Atender Paciente

Una vez ya distribuidos los pacientes en sus respectivos diccionarios (dependiendo de la especialidad pedida y el grado de urgencia asignado), cuando un doctor se desocupa, dependiendo de su especialidad, se le asigna un nuevo paciente. Para obtener la especialidad de un doctor, debemos *obtener* el valor asociado a su clave en el ABB donde almacenamos a todos los doctores del sistema; obtener en un ABB cuesta *O(log d)* -siendo d la cantidad total de doctores-.

Como la prioridad la tienen los pacientes urgentes encolados en la especialidad del doctor ya obtenida, primero se accede a los pacientes en el Hash de los pacientes urgentes. Si la cola asociada a la clave de la especialidad buscada tiene algún paciente encolado, se desencola, lo cual tiene una complejidad de *O(1)*. Esto fue logrado, dado que modificamos el TDA Cola. Agregamos un atributo extra, el cual, dependiendo de si se encola o se desencola, actualiza la cantidad de elementos en la cola. Esto lo hace más eficiente y nos ahorra tener que desencolar todos los elementos, utilizar un contador y luego volver a dejar la cola en su estado original. Por lo tanto, atender paciente en casos urgentes terminará teniendo una complejidad temporal de *O(log d) + O(1) = O(log d)*. En el caso de que la cola asociada a la clave de la especialidad esté vacía (no hay pacientes urgentes), se

proseguirá a buscar en el segundo Hash (el de los pacientes regulares). En este caso, como se debe desencolar de un Heap (cuya complejidad es *O(log n),* siendo n la cantidad de pacientes en esa especialidad), la complejidad temporal total será de O(*log d)* + O(*log n)*.

Informe doctores

Para realizar la última parte de este TP, decidimos agregar al TDA ABB una primitiva que permite iterar por rangos. El objetivo de la primitiva es que, si la cantidad de elementos dentro de ese rango es menor a la cantidad total de elementos, la complejidad temporal de la misma será de *O(log d)*. En el caso en que todos los elementos del ABB formen parte del rango introducido por el usuario, se ejecutará en *O(d)* porque estaría visitando a todos los nodos del árbol. También agregamos un contador que devuelva el número de elementos dentro del rango pedido a la hora de iterar.

Para la primitiva decidimos implementar un recorrido *inorder* (primero visitar a mi hijo izquierdo, luego a mí mismo, y luego a mi hijo derecho) para que los elementos se visiten en orden alfabético tal como pide el enunciado.

Diseñamos una función *visitar* (para usar en la nueva primitiva de iteración por rangos) que permite imprimir por pantalla lo pedido a la hora de presentar el informe de doctores, para que no necesitemos acceder a los elementos del ABB una vez terminada la iteración (lo cual aumentaría la complejidad).