Trabajo Práctico 2

Informe

Grupo: 25

Integrantes: Joaquin Lerer y Dante Reinaduo

Ayudante: Sofia Morsaletto

Análisis y diseño de la solución

Luego de analizar durante varios días la consigna del trabajo práctico, sus requisitos y condiciones, optamos por diseñar un TDA Clínica cuyas primitivas nos ayudasen con los comandos solicitados. De esta manera, nuestro trabajo quedó conformado por el programa principal **zyxcba.c**, en el cual se crea la clínica pasándole por parámetro el nombre de los archivos de doctores y pacientes. Luego, mediante una función que procesa los comandos, hace uso de las primitivas de este TDA para llevar a cabo las acciones.

La estructura principal del TDA Clínica es la siguiente :

```
struct clinica{
     abb_t* doctores;
     hash_t* pacientes;
     hash_t* especialidades;
}
```

Como puede observarse, este TDA emplea otros tipos de datos abstractos en su funcionamiento, tales como el hash y el abb. A su vez, los datos almacenados por estos, también son estructuras las cuales fueron definidas convenientemente para la realización del trabajo práctico.

Árbol de doctores: Este miembro de la clínica es un árbol binario de búsqueda (ABB) cuya clave es el nombre del doctor y su dato es un puntero a una estructura denominada doctor, conformada de la siguiente manera:

```
struct doctor{
     char* nombre,
     char* especialidad;
     int pacientes_atendidos;
}
```

Este ABB es generado cuando se crea la clínica, utilizando una función que recibe por parámetro el nombre del archivo con la información de los doctores y lo procesa. Por cada línea procesada del archivo se crea un struct doctor, donde se almacena su nombre y especialidad. Luego, se almacena en el ABB el nombre del doctor como clave y el puntero a la estructura como dato.

Por otro lado, dado que en el informe se requiere mostrar la cantidad de pacientes atendidos por doctor, asignamos un miembro `pacientes_atendidos` el cual es inicializado en 0 en la creación de la estructura.

Es importante aclarar, que en la creación de este ABB se pasa como función de comparación a strcmp para que al realizar un recorrido in order, se recorran los nombres de los doctores en orden alfabético. También se pasa una función de destrucción acorde a la estructura almacenada como dato.

Decidimos utilizar un ABB para almacenar a los doctores, dado que, a diferencia de un hash, nos permite ordenar la información, lo cual es muy útil teniendo en cuenta el tercer comando. Además, siendo d la cantidad total de doctores en la clínica, dado que el ABB es un estructura recursiva, las búsquedas se realizan en O(log(d)), lo cual es acorde al orden requerido por el trabajo práctico.

Hash de pacientes: Este miembro de la clínica, es un hash cuya clave es el nombre del paciente y su dato es su año de ingreso a la clínica. Si bien realizamos una estructura paciente, no nos pareció necesaria emplearla dentro del hash y para optimizar la memoria utilizada, solo será creada cuando se la necesite.

Este hash es generado cuando se crea la clínica, utilizando una función que recibe por parámetro el nombre del archivo con la información de los pacientes y lo procesa. Por cada línea procesada, se valida que el año de ingreso a la clínica del

paciente sea un número válido, en caso de no serlo, se imprime un mensaje de error, la función devuelve NULL, y el programa principal aborta con código numérico 1, liberando toda la memoria solicitada.

Decidimos utilizar un hash para almacenar a los pacientes, dado que las búsquedas son aproximadamente O(1), lo cual es muy eficiente y agiliza mucho el flujo del código. Además, teniendo en cuenta los comandos solicitados, no consideramos necesario contar con un orden relativo entre los pacientes, a diferencia de lo que sucede con los doctores, por lo que un hash nos pareció la solución más óptima.

Hash especialidades: Este miembro de la clínica es fundamental para poder llevar a cabo el primer y segundo comando solicitados en el presente trabajo práctico. Se trata de un hash cuya clave es el nombre de la especialidad y su dato es un puntero a una estructura denominada especialidad, conformada de la siguiente manera:

```
struct especialidad{
     char* nombre;
     cola_t* doctores;
     cola_t* urgencias;
     heap_t* regulares;
     int cant_pacientes;
}
```

Este hash es generado cuando se crea la clínica, con el fin de ahorrarnos tener que recorrer todos los doctores dos veces, el hash se crea en la misma función que procesa el archivo con la información de los doctores. Como podrá observarse en nuestro código, dicha función recibe, además del nombre del archivo, un puntero a un hash. Por cada línea procesada, se verifica si la especialidad del doctor pertenece al hash, en caso de no hacerlo, se crea una estructura especialidad, y se le asigna el nombre correspondiente. Luego, se encola al doctor en el miembro

doctores de dicha estructura, y se almacena en el hash. En caso de que la especialidad ya perteneciera al hash, simplemente se procede encolando al doctor en el miembro doctores de la estructura correspondiente.
Por otro lado, dado que en el segundo comando se requiere mostrar la cantidad de pacientes en lista de espera para cierta especialidad, asignamos a la estructura un miembro `cant_pacientes` el cual es inicializado en 0 durante su creación.
Decidimos utilizar un hash para almacenar a las especialidades, dado que las búsquedas son aproximadamente O(1), lo cual es muy eficiente y dado que debemos trabajar bastante con los datos almacenados, es decir, con la estructuras especialidades, poder acceder a estas desde un hash optimiza mucho el código. Además, teniendo en cuenta los comandos solicitados, como pasó en el caso los pacientes, no consideramos necesario contar con un orden relativo entre las especialidades, por lo que un hash nos pareció la solución más óptima.

Una vez explicada la estructura principal y lógica de nuestro programa, queremos hacer algunos comentarios con respecto a su distribución. Como fue mencionado anteriormente, nuestro código está compuesto por un programa principal **zyxcba.c**, el cual hace uso de las primitivas del TDA Clínica para llevar a cabo las acciones solicitadas, el cual , a su vez, hace uso de diversos tipos de datos abstractos (tales como: ABB, Heap, Hash y Cola) . Con el fin de modularizar nuestro código y facilitar el proceso de *debugging*, optamos por fragmentar diversas funciones del TDA Clínica en varios archivos. Por esta razón, tendremos un archivo principal **clínica.c**, con su correspondiente **clinica.h**, donde se encontraran todas las primitivas, y otros archivos auxiliares que ayudarán a su realización. Por lo tanto, tendremos un archivo **csv.c** donde se encuentran las funciones relacionadas al

procesamiento de archivos y otro archivo **structs.c** , donde están definidas las estructuras doctor, paciente y especialidad, y todos las funciones relacionadas a estas. Una vez aclarado esto, pasaremos a analizar el diseño y solución de los tres comandos solicitados.

Análisis y diseño de los comandos

Una vez iniciado el programa, haber recibido los archivos correspondientes, y suponiendo que no hubo ningún problema durante la creación de nuestro TDA, la clínica se habrá generado correctamente y nuestro programa estará listo a la espera de instrucciones.

1) Primer Comando: Pedir Turno

Una vez procesado la entrada con una función acorde y haber identificado el comando, deberíamos obtener tres parámetros: nombre del paciente, especialidad y grado de urgencia. Estos tres parámetros deben ser validados, como contamos con un hash de especialidades y otro de pacientes, saber si un elemento pertenece a un hash se realiza en tiempo O(1). Por otro lado, el grado de urgencia, se valida comparando el parámetro recibido con los dos casos posibles: URGENTE o REGULAR, cada comparación se lleva a cabo en tiempo O(1).

Una vez validado los parámetros, obtendremos del hash especialidades el dato asociado a la especialidad pedida, esta accion tambien se realiza en tiempo O(1).

En caso de que el grado de urgencia del paciente sea urgente, se procederá a encolar al paciente en cola urgencias de la estructura especialidad y se aumentará en una unidad el miembro `cant_pacientes`. Luego, se imprime por pantalla el mensaje solicitado. Si nos ponemos a analizar, podemos observar que tanto encolar en una cola, realizar una suma y imprimir por pantalla un mensaje, son operaciones que se ejecutan en O(1). Si sumamos el costo de todas las operaciones realizadas hasta el momento, obtendremos que el precio de pedir turno en un caso urgente se lleva a cabo en O(1), tal como se solicita en el enunciado del trabajo práctico.

Por otro lado, en el caso de que el grado de urgencia sea regular, se procederá a obtener en el hash de pacientes, el año de ingreso a la clínica, que como se trata de un hash, se lleva a cabo en tiempo O(1). Luego, se procederá a crear la estructura paciente, cuyo formato es el siguiente:

```
struct paciente{
     char* nombre,
     int anio;
}
```

Una vez creada esta estructura, que cuenta con el nombre del paciente y su año de ingreso a la clínica, es encolada en el heap regulares de la estructura especialidad y se aumenta en una unidad el miembro `cant_pacientes`. Es importante aclarar que, este es un heap de mínimos, el cual cuenta con una función de comparación que compara y ordena según el año de ingreso a la clínica de cada paciente. Por lo tanto los pacientes con más antigüedad en la clínica, tendrán mayor prioridad. Una vez realizado esto, se imprime por pantalla el mensaje solicitado. Si nos ponemos

a analizar, el costo de realizar la validación de los parámetros, obtener el dato en el hash de especialidades, hacer una suma e imprimir un mensaje todas son operaciones que se ejecutan en tiempo O(1). A fines prácticos, se puede considerar que el precio de crear la estructura paciente, también, es O(1). Luego, siendo n la cantidad de pacientes almacenados dentro de un heap, el costo de encolar otro paciente, en el peor de los casos, es O(log(n)), debido a la utilización de Upheap. Si sumamos el costo de todas las operaciones realizadas hasta el momento, obtendremos que el precio de pedir turno en un caso regular se lleva a cabo en O(log(n)) + O(1) lo que equivale a O(log(n)), tal como se solicita en el enunciado del trabajo práctico.

2) Segundo Comando: Atender siguiente paciente

Una vez procesado la entrada con una función acorde y haber identificado el comando, deberíamos obtener un único parámetro: nombre del doctor. Este parámetro debe ser validado, como contamos con un ABB de doctores, siendo d la cantidad de doctores en el sistema, validar si el doctor pertenece se lleva a cabo en tiempo O(log(d)). Luego, obtenemos el dato asociado a esta clave, es decir, la estructura doctor, accion que también demora O(log(d)). Una vez hallada la especialidad que ejerce dicho doctor, obtendremos del hash especialidades el dato asociado a esta, acción que se realiza en tiempo O(1).

En el caso de que haya pacientes en la lista de espera con grado urgente, se procederá a desencolar un elemento de la cola urgencias de la estructura especialidad. Luego, se añadirá en una unidad la cantidad de pacientes atendidos por dicho doctor y se disminuirá en una unidad la cantidad de pacientes en la espera para la especialidad. Finalmente, se imprime por pantalla el mensaje

requerido. Si nos ponemos a analizar, podemos observar que tanto desencolar en una cola, realizar una suma, realizar un resta e imprimir por pantalla un mensaje, son operaciones que se ejecutan en O(1). Sumando el costo de todas las operaciones realizadas hasta el momento, obtendremos que el precio de atender al siguiente paciente en estado de urgencia se lleva a cabo en 2*O(log(d)) + O(1), lo que es equivalente a O(log(d)) tal como se solicita en el enunciado del trabajo práctico.

Por otro lado, en el caso que no haya pacientes en estado de urgencia en la lista de espera, pero si pacientes regulares, se procederá a desencolar un elemento del heap de regulares dentro de la estructura especialidad. Tal como estudiamos durante la cursada, el costo de desencolar en un heap es O(log(n)), siendo n la cantidad de elementos almacenados en el heap. Luego, se añade en una unidad la cantidad de pacientes atendidos por dicho doctor y se disminuye en una unidad la cantidad de pacientes en la espera para la especialidad. Finalmente, se imprime por pantalla el mensaje requerido. Si nos ponemos a analizar, podemos observar que tanto realizar una suma, realizar una resta e imprimir por pantalla un mensaje, son operaciones que se ejecutan en O(1). Sumando el costo de todas las operaciones realizadas hasta el momento, obtendremos que el precio de atender al siguiente paciente en estado regular se lleva a cabo en O(log(d)) + O(log(n)) + O(1), lo que es equivalente a O(log(d) + log(n)) tal como se solicita en el enunciado del trabajo práctico.

En caso de que no hubiera pacientes en la lista de espera, simplemente se imprime por pantalla que no hay pacientes en espera para dicha especialidad.

3) Tercer Comando: Informe doctores

Por último, pero no menos importante, al procesar este comando deberíamos obtener dos parámetros, el nombre del doctor desde el que se desea iniciar el informe y el nombre del doctor donde se lo quiere finalizar. Además, en caso de no quererse indicar un inicio o fin, simplemente se pasa un parámetro vacío. Para poder efectuar este comando, tuvimos que añadir una primitiva al ABB que nos permitiera iterar in order y por rangos, dado que los iteradores con los que contábamos comenzaban a iterar desde el mínimo, por lo que, utilizarlos resultaría en una complejidad mayor a la que se pide en el enunciado. A su vez, esta nueva primitiva es un iterador interno, que recibe por parámetro las claves que delimitan el rango de iteración y una función callback "visitar" que recibe la clave, el valor y un puntero extra, y devuelve true si se debe seguir iterando o false en caso contrario.

Como fue comentado anteriormente, siendo d la cantidad de doctores en el sistema, validar si los doctores pertenecen al ABB doctores se lleva a cabo en tiempo O(log(d)). Si se desea mostrar un número N de doctores, utilizamos la nueva primitiva, pasandole una función que añade los N doctores a una lista, y dicha lista como extra. El costo de realizar esta última operación es N veces O(1), lo que es equivalente a O(N). Luego, se imprime la cantidad de doctores que van a ser mostrados en el informe, acción que vale O(1). Finalmente, se recorre la lista y por cada doctor se imprime el mensaje solicitado, dado que recorremos los N doctores, efectuar esto nos lleva un tiempo O(N). Sumando el costo de todas las operaciones realizadas hasta el momento, obtendremos que el precio de realizar el informe de doctores es O(log(d)) + O(N). Analizemos el orden obtenido, si se desea realizar un informe de todos los doctores en la clínica, entonces la cantidad N es igual a d,

por lo que obtenemos un orden de O(log(d) + d), el cual equivale a un orden O(d), cumpliendo con las condiciones requeridas por el trabajo práctico. Por otro lado, si se desea mostrar una cantidad de doctores considerablemente menor al total, de tal manera que $N <<< \log(d)$, podemos considerar a N despreciable ante log(d), y el orden se puede aproximar por O(log(d)), cumpliendo así con las condiciones requeridas por el trabajo práctico. En general, para los casos restantes, en los que el número de doctores a mostrar es menor a d, pero no despreciable ante este, el costo de realizar el informes es O(log(d) + N).