TP N°2

Sistema de gestión de la clínica zyxcba

Algoritmos y Programación II 75.41

Cátedra Buchwald

Alumnos:

Nombre: Franco Ariel Valfré

Padrón: 105607

Nombre: Bruno Bevilacqua

Padrón: 105534

Corrector: Agustín López Núñez

Objetivo

Diseñar e implementar en lenguaje C el sistema de gestión de turnos y generación de informes de la clínica zyxcba. La interacción con el sistema se realiza mediante comandos ingresados por entrada estándar, siendo éstos pedir turno, atender siguiente y informe.

Condiciones

La gestión de turnos debe respetar un sistema de prioridades específico, en primer lugar el sistema debe asignar prioridad máxima a los pacientes ingresados en urgencia, y en segundo lugar, cuando los turnos ingresados sean regulares, asignar más o menos prioridad en base a la antigüedad del paciente en la clínica.

Para la atención de pacientes, se deberá almacenar la cantidad de pacientes atendidos por cada doctor, que luego es necesaria para la generación de informes.

El informe se puede generar sobre un doctor en específico, sobre un rango fijo de doctores o sobre todos los almacenados en el sistema.

Problemas principales

1. La forma de almacenar en el sistema la información de todos los doctores anotados en la clínica junto con sus especialidades y todos los pacientes anotados en la clínica junto con sus antigüedades, ya que se busca que la ejecución de todos los comandos sean lo más “rápida” posible.
2. La manera de manejar los turnos junto con sus prioridades específicas, como almacenarlos y dividirlos para cada especialidad.
3. Los comandos deben cumplir con una complejidad temporal deseada:
    a. En el caso de pedir turno es O(1) en un caso urgente o O(log n) en un caso regular.
    b. Para atender siguiente es O(log d) en urgencias o O(log d + log n) en caso regular.
    c. En el informe de doctores O(d) en el peor caso y O(log d) en un caso promedio

    Para todas las complejidades dadas n es la cantidad de pacientes en una especialidad dada y d la cantidad de doctores en el sistema

Resolución

Carga de datos

Pacientes:

Es deseable poder chequear rápidamente si un paciente existe o no en el sistema, y de existir poder conocer su antigüedad. La estructura de datos que más se adecúa a las necesidades de este problema es un hash, debido a que sus (métodos?) tienen complejidad O(1), por ende, preguntar si existe un paciente en el sistema o acceder a su antigüedad cuando existe se ejecutan en tiempo constante . Por lo tanto se eligió un hash y los datos se almacenaron con el nombre del paciente como clave y su antigüedad como valor asignado.

Doctores:

Se necesita cumplir distintos valores(sacar), en primer lugar es necesario poder almacenar a los doctores junto con sus especialidades y que poder chequear si un doctor está o no listado en el sistema rápidamente, además de esto hay que mantener la cantidad de pacientes atendidos por cada doctor, y finalmente se debe poder generar el informe de doctores con la información buscada cumpliendo la complejidad deseada de O(d) en caso de pedir un informe de todos los doctores o O(log d) en un caso promedio.

La estructura de datos que puede cumplir con todas esas condiciones es un abb (árbol binario de búsqueda), debido a que:

1. Preguntar si una clave existe en el abb se realiza en tiempo logarítmico (O(log n), siendo n la cantidad de claves almacenadas), que si bien es peor que tiempo constante sigue siendo mucho mejor que algo en tiempo lineal.
2. Las claves deben mantener un orden alfabético para generar el informe.
3. El abb permite (con algunas modificaciones) iterar en un rango establecido sin necesidad de recorrer todas las claves, esto invalida al hash como posible opción, debido a que sería imposible dado que el hash no mantiene ordenadas las claves almacenadas al contrario de abb, debido a que este último mantiene la estructura ordenada en base a su propiedad que siempre cumple, siendo que el nodo “padre” es mayor a su “hijo” izquierdo y menor que su “hijo” derecho.

Por lo tanto se eligió un abb como estructura para almacenar la información de los doctores, almacenando el nombre del doctor como clave y un struct que contiene su especialidad y el contador de pacientes atendidos. Las claves mantienen un orden alfabético que permiten luego generar el informe.

Listas de espera:

Las listas de espera de cada especialidad deben ser independientes, dado que un paciente puede pedir turnos para más de una especialidad, y cada doctor tiene una especialidad específica y además es necesario que se pueda acceder a dichas listas de espera rápidamente para no perder tiempo siempre buscando dónde está cada lista de espera.

Debido a que se busca rapidez la mejor estructura de datos que se adecúa al problema es un hash, dado que como se vio anteriormente sus operaciones se realizan en tiempo constante, y al atender un paciente siempre se tiene la información de la especialidad, dado que el doctor la contiene. Por lo tanto, se almacenan en el hash el nombre de la especialidad como clave y las listas de espera ligadas a cada una como valores.

Manejo de turnos y prioridades

Debido a que existe una prioridad para ordenar los turnos regulares y otra distinta para los turnos urgentes, no basta solamente una cola de prioridad o heap para resolver el problema, ya que estos se manejan con un único tipo de prioridad para comparar los elementos guardados. Por lo tanto, era necesaria la implementación de un nuevo TDA que pueda manejar ambas prioridades a la vez.

Este TDA, denominado “cola_vip”, necesitaba cumplir con las complejidades temporales deseadas:

1. O(1) para encolar/desencolar un paciente urgente y que estos salgan en orden de llegada.
2. O(log n) para encolar/desencolar un paciente regular, y que estos salgan en base a sus antigüedades, los más antiguos primero.

Para cumplir con estas condiciones se decidió utilizar una cola para encolar los pacientes urgentes, debido a que las operaciones encolar/desencolar se cumplen en tiempo constante y los datos salen en base al orden de llegada, y un heap para los pacientes regulares, debido a que encolar/desencolar tienen complejidad O(log n), siendo n la cantidad de datos almacenados, y los datos salen en base a la prioridad asignada.

Este TDA tiene además las operaciones normales de una cola o un heap, tal como ver el primer elemento, preguntar si está vacía o pedir la cantidad de elementos almacenados.

Comando pedir turno

Luego de validar que los parámetros ingresados sean correctos, simplemente se obtiene la lista de espera de la especialidad dada y se encola al paciente.

Las complejidades pedidas se cumplen debido a que:

1. En el caso de urgencias, “cola_vip” almacena el valor en la cola común, tardando O(1).
2. En el caso regular, “cola_vip” almacena el valor en el heap, y debido a las operaciones realizadas por él para mantener el orden lo realiza en O(log n), siendo n la cantidad de pacientes regulares encolados.

Comando atender siguiente

Luego de validar los parámetros se obtiene el doctor junto con su especialidad y cantidad de pacientes atendidos, con la especialidad se busca la lista de espera correspondiente, se desencola el siguiente paciente y se actualiza la cantidad atendida.

Las complejidades pedidas se cumple debido a que:

1. En el caso de urgencias, obtener el doctor junto con su información se realiza en O(log d), con d siendo la cantidad de doctores en el sistema, debido a que se buscan en el abb de doctores, y desencolar el siguiente paciente se ejecuta en O(1) por desencolar un paciente urgente de “cola_vip”.
2. En el caso regular, la parte del doctor es idéntica, pero en este caso desencolar el siguiente paciente cuesta O(log n), siendo n la cantidad de pacientes en la especialidad, debido a “cola_vip”.

Comando informe

Para que el comando funcione primero se debieron realizar unas modificaciones al abb, más específicamente al iterador in order del mismo.

Las modificaciones se centran en el movimiento entre nodos del árbol, debido a que siempre hay que tener en cuenta los límites indicados para no salirse de rango establecido al avanzar, y además lidiar con las distintas combinaciones de límites posibles, ya que existen casos donde hay que recorrer todo el abb, o desde un inicio concreto hasta el final, o viceversa, o con doctores que no existen en el sistema…

Dado que las modificaciones hechas se basan en validaciones/chequeos, no tienen un peso mayor en la complejidad, por lo que la complejidad se basará en la cantidad de

vértices que se visiten en cada iteración, por lo que las complejidades se cumplen debido a que:

1. Al no especificar ningún límite, el iterador simplemente “visitará” cada nodo del abb “in order”, por lo que la complejidad será O(d), siendo d la cantidad de doctores en el sistema
2. Al existir límites, el iterador “visitará” los nodos que estén dentro de ese rango, variará dependiendo de la cantidad pero en promedio lo hará en O(log d), siendo d la cantidad de doctores en el sistema