

Trabajo práctico N°2

Algoritmos y Programación 2 Cátedra Buchwald

Primer cuatrimestre 2020

Barberis, Juan C. (105147)
Gabrielli, Agustín (104954)

Corrector: Collinet, Jorge

Análisis y diseño

Para poder cumplir con la complejidad temporal que nos exigió la cátedra, decidimos implementar el sistema de la clínica utilizando múltiples estructuras de datos vistas en la cursada, concretamente el hash, el árbol binario de búsqueda (abb), la cola de prioridad implementada como heap, la pila y la cola. A continuación, se explicará el por qué de la implementación y de las soluciones llevadas para cada parte del programa.

Inicio del programa

Cuando nuestro programa inicia, recibimos dos archivos con la información de los doctores y los pacientes que pertenecen a la clínica. Estos datos llegan en un formato de archivo CSV (valores separados por coma) y son parseados a una estructura de datos por un módulo brindado por la cátedra. Además de estos archivos, tenemos que cargar las especialidades que los doctores pueden atender en la clínica.

Para almacenar estos datos en memoria, utilizamos tres estructuras: ABB para doctores, Hash para pacientes y Hash para especialidades.

¿Por qué un ABB para doctores? Por la eficiencia de búsqueda por orden alfabético que nos otorga esta estructura a la hora de precisar un listado (por rangos) de los doctores en la clínica.

¿Por qué un Hash para pacientes? Para los pacientes, utilizar un Hash fue la decisión definitiva. El acceso a la estructura de cada uno en un tiempo constantes nos facilita mucho para, por ejemplo, procesar un turno.

¿Por qué un Hash para las especialidades? Porque al igual que con los pacientes, necesitamos la mayor velocidad para trabajar sobre una especialidad. Esta, por dentro cuenta con dos sub-estructuras: cola y heap. Cola para pacientes urgentes y Heap para pacientes regulares que son atendidos por antigüedad en la clínica. *Esto se detalla mejor en los siguientes párrafos.*

Una vez que realizamos las validaciones y cargamos los datos en memoria, el programa estará listo para recibir instrucciones del usuario desde la terminal.

Operaciones

En base al siguiente análisis de las operaciones que puede hacer el usuario decidimos implementar la estructura de datos que se explicó anteriormente. La justificación detallada de la elección de ellas se hará a continuación. Hay tres comandos posibles:

1) Pedir turno. Cuando se pide un turno para:

- un paciente *urgente*, se lo encola en la cola de esa especialidad, que se obtiene a través del hash de especialidades en tiempo constante. Esto es porque los urgentes se resuelven por orden de llegada (y la cola, al ser una estructura fifo nos brinda esa utilidad). Además, encolar de una cola es $O(1)$ como se pide. Entonces, tendremos una cola por cada especialidad.
- un paciente *regular*, se lo encola en el heap (de máximos, con la antigüedad en la clínica como prioridad) de dicha especialidad (también se obtiene ese heap a través del hash de especialidades). O sea, tendremos un heap por cada especialidad. Como hay p pacientes en esa especialidad, encolar sería $O(\log p)$ como nos piden.

Además, para los pacientes urgentes no podemos tener una sola cola por razones obvias: supongamos que encolamos a un paciente para la especialidad A en una cola genérica y luego encolamos a otro paciente en la misma para la especialidad B; si después atendemos al siguiente paciente para un doctor en la especialidad B, va a desencolar al que estaba para la especialidad A, y eso es incorrecto.

Por otro lado, con respecto a los pacientes regulares, se podría tener a todos los pacientes con turno en un heap genérico (se puede tener repetidos). El problema radica en que desencolar de ese heap es $O(\log n)$, siendo n la cantidad total de pacientes de todas las especialidades y en los requerimientos del TP se pide $O(\log p)$ siendo p la cantidad de pacientes con turno para *esa* especialidad. Con lo cual debemos tener tantos heaps como especialidades.

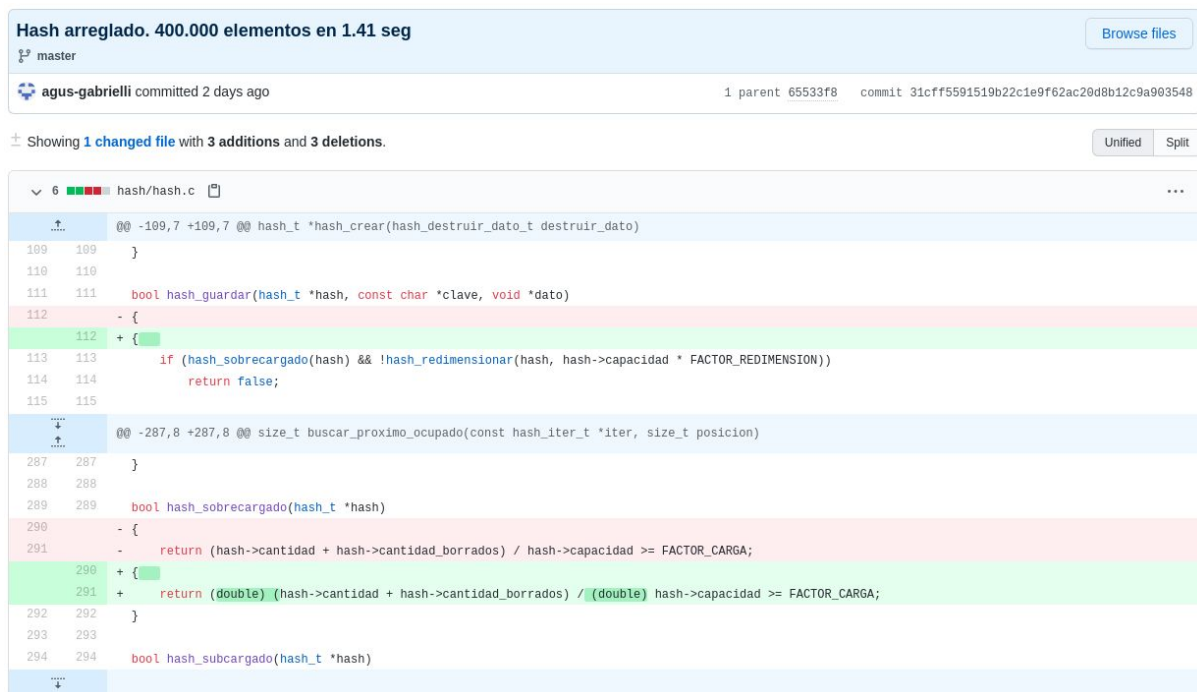
2) Atender siguiente paciente. Se recibe al doctor que está libre. Como cada doctor tiene una sola especialidad, se procede a obtener la especialidad de ese doctor del `abb`, esto es $O(\log d)$. Ya con la especialidad en mano, a través del hash de especialidades ($O(1)$) vamos a la cola de urgencia y desencolamos en $O(1)$. El total de la operación es $O(\log d)$. Si no hay

pacientes urgentes, entonces se tendrá que desencolar del heap de pacientes con turno regular en esa especialidad, lo cual es $O(\log p)$. El total quedaría $O(\log d + \log p)$, tal como se pide.

3) Informe doctores. Es acá donde surge la utilidad del abb por sobre una implementación con hash. Como tenemos la información de los doctores en el ABB, lo único que queda por hacer es iterar por rangos, in order. Para llevar a cabo esto, hay que implementar esta primitiva en el abb. La complejidad de esta operación, implementada de esta forma, cumple con lo pedido: recorrer todo el ABB es $O(d)$ siendo d la cantidad de doctores, pero en promedio, iterar por rangos no muy grandes es logarítmico.

Dificultades

La única dificultad que se nos presentó fue al finalizar la implementación, cuando corrimos las pruebas y notamos que uno de nuestros TDA estaba fallando. Después de un extenso análisis, llegamos a la conclusión de que había un error en un casteo del TDA Hash, donde la redimensión nunca ocurría.



```
Hash arreglado. 400.000 elementos en 1.41 seg
master
agus-gabrielli committed 2 days ago
1 parent 65533f8 commit 31cff5591519b22c1e9f62ac20d8b12c9a903548
Showing 1 changed file with 3 additions and 3 deletions.
hash/hash.c
@@ -109,7 +109,7 @@ hash_t *hash_crear(hash_destruir_dato_t destruir_dato)
109 109 }
110 110
111 111 bool hash_guardar(hash_t *hash, const char *clave, void *dato)
112 - {
112 + {
113 113     if (hash_sobrecargado(hash) && !hash_redimensionar(hash, hash->capacidad * FACTOR_REDIMENSION))
114 114         return false;
115 115
@@ -287,8 +287,8 @@ size_t buscar_proximo_ocupado(const hash_iter_t *iter, size_t posicion)
287 287 }
288 288
289 289 bool hash_sobrecargado(hash_t *hash)
290 - {
290 + {
291 -     return (hash->cantidad + hash->cantidad_borrados) / hash->capacidad >= FACTOR_CARGA;
291 +     return ((double) (hash->cantidad + hash->cantidad_borrados) / ((double) hash->capacidad >= FACTOR_CARGA;
292 292 }
293 293
294 294 bool hash_subcargado(hash_t *hash)
```

Commit del problema solucionado. Solo bastó con realizar un casteo a double.

Conclusiones

Más allá del error anteriormente mencionado, nos basamos en una planificación extensa antes de comenzar a programar. Nos aseguramos que los planteos sean lógicos y que estén revisados por ambos. Este método hizo sencilla y rápida la implementación.