Trabajo práctico N°2

Algoritmos y Programación II
Curso Buchwald (ex Wachenchauzer)

Integrantes:

Proz, Gastón Leandro

Mail: gproz@fi.uba.ar

Padrón: 105868

Villegas Cabral, Tomas Ariel

Mail: tvillegas@fi.uba.ar

Padrón: 106456

Corrector:

Raik, Ignacio

Índice:

1. Introducción al trabajo práctico:

1.1. Introducción:

En Islandia, la clínica Zyxcba Inc. organiza sus listas de espera a partir del año de inscripción como paciente. Así, dentro de cada especialidad se da prioridad a aquellos pacientes que se hayan inscripto hace más tiempo, pero también considerando la urgencia de la consulta. Para celebrar el 75 aniversario de la clínica, Zyxcba Inc. ha decidido remplazar la implementación actual del sistema de gestión en COBOL por una más moderna en C.

1.2. Consigna:

Se pide implementar el sistema de gestión de turnos y generación de informes de la clínica.
Cuando el sistema arranca, carga en memoria la lista de pacientes y la lista de les doctores. A partir de ese momento, lee de entrada estándar las operaciones a realizar (una por línea), tal y como se detalla a continuación.
Tras cada operación, el sistema siempre imprime por salida estándar un mensaje, informando de los resultados, o cualquier error que haya ocurrido.

1.3. Objetivos y comentarios sobre el trabajo práctico:

La finalidad del trabajo práctico es ver el potencial que poseen los TDAs implementados anteriormente, como se indica en su nombre, son abstractos y este TP es una forma de verlo. A diferencia del primer trabajo práctico, en este caso tuvimos que pensar que estructura se adaptaría mejor a los problemas planteados, es decir, cumplir con las complejidades temporales pedidas y además que las nuevas estructuras tengan sentido. También nos sirvió para ver los usos que tienen las estructuras tales como el Hash (Aunque lo conocíamos como diccionario), Heap y el ABB, estructuras que cumplen la función de almacenar datos, pero el cómo lo hacen es lo importante y encontrar formas y/o ver en que casos utilizarlos, es lo que nos llevamos de este TP.

2. Diseño de soluciones:

2.1 Estructuras de datos creadas especialmente para el trabajo:

Primero, nos gustaría dejar por sentado las tres nuevas estructuras de datos que utilizaremos a lo largo del siguiente informe.

La **primera** nueva estructura es la del paciente, la finalidad es ser guardada en un Hash para simular el registro de pacientes de una clínica, el Hash en particular tendrá como claves los nombres de los pacientes y como dato esta nueva estructura. Quizá pueda parecer algo repetitivo utilizar la clave como nombre y luego que la estructura también contenga el nombre del paciente, pero la finalidad del nombre no es para utilizarlo en el Hash sino en un Heap, para saber el nombre del paciente desencolado.

```
typedef struct paciente {
    char* nombre;
    long int anio;
}paciente_t;
```

La **segunda** estructura utilizada es la del doctor, la idea de esta estructura es utilizarla en un **ABB**, así podremos cumplir más adelante con las complejidades temporales deseadas. El **ABB** tendrá como claves los nombres de los doctores y como dato esta estructura, de igual forma que antes, la explicación del porqué está el nombre del doctor en la estructura, es porque es necesaria para la creación del **ABB.**

```
typedef struct doctor{
    char* nombre;
    char* especialidad;
    size_t atendidos;
}doctor_t;
```

La **tercera** estructura utilizada es la de las especialidades, inicialmente se crean a la par junto con la de los doctores, la diferencia es que las especialidades se guardan en un Hash, donde la clave es el nombre de la especialidad y el dato es la estructura mostrada a continuación. Cabe destacar que el Heap que utilizamos es de mínimos, debido a que queremos priorizar a los pacientes más antiguos.

```
typedef struct especialidad {
    cola_t* pacientes_urgentes;
    heap_t* pacientes_regulares;
    size_t pacientes;
}especialidad_t;
```

2.2 Lectura de los archivos **.csv**:

Se utilizó la librería brindada por la cátedra para la lectura y parseo de archivos **.csv**, gracias a esto y las funciones **constructoras de datos** que pasamos por parámetro, pudimos crear las estructuras principales del TP, es decir, los pacientes, doctores y especialidades del sistema. Las primitivas que se encargan de esto son:

- `void* crear_paciente(char** argumentos, void* extra)`
- `void* crear_doctor(char** argumentos, void* extra)`

Ambas funciones realizan acciones similares, pero la diferencia es que en la primitiva `crear_doctor` se crean especialidades también, debido que al leer la información del **.csv**, también se obtiene información sobre el área en la cual se desempeñan diariamente. Estas funciones constructoras funcionan en O(1) si se toma como constante el tiempo que se toma en pedir la memoria dinámica.

3. Comandos implementados:

3.1. Pedir turno:

Como estructura principal se decidió usar un Hash de especialidades, esto nos permite en O(1) acceder a la información de dicha especialidad.

- Si se tratase de un **caso urgente**, deberá funcionar en O(1), para ello se decidió utilizar una cola normal que permite encolar y desencolar en el tiempo requerido, además al ser una estructura FIFO, los pacientes se registran por orden de llegada.
- Si se tratase de un **caso regular**, puede funcionar en O(log n), siendo n la cantidad de pacientes encolados en la **especialidad requerida**, para ello se decidió utilizar un Heap que toma como prioridad a los pacientes con mayor antigüedad en la clínica.
  Esta estructura nos permite encolar en O(log n) tal como se pide en la consigna.

3.2. Atender siguiente:

Como estructuras se utilizan un ABB de doctores y el Hash de especialidades, el parámetro que acompaña al comando es el nombre del doctor que quedó libre, el ABB permite acceder a la información del doctor en O(log d), siendo d la cantidad de doctores que se encuentran en el sistema, dentro de la información de dicho doctor se encuentra la especialidad a la que pertenece, con esa información se puede acceder a las colas de espera del Hash de especialidades en O(1), lo que no aumentaría la complejidad. Siempre se debe atender primero a los casos urgentes y en caso de que no haya más casos urgentes se atiende al paciente regular con más prioridad, de la especialidad correspondiente al doctor.

- Si se atiende a un **paciente urgente**, como la cola utilizada es una normal, la complejidad al desencolar es O(1), como resultado la complejidad total es O(log d).

- Si se atiende a un **paciente regular**, como se utiliza un Heap, la complejidad al desencolar es O(log n), siendo **n** la cantidad de pacientes restantes en la **especialidad correspondiente al doctor**, como resultado la complejidad total es O(log d + log n).

Al ser atendido un paciente, se incrementa en uno la cantidad total de pacientes atendidos por el doctor, realizar esta acción es O(1).

3.3. Informe de doctores:

Como estructura principal se utilizó el ABB de doctores, la razón principal por la cual se eligió esta estructura por sobre las demás, fue para poder cumplir tanto la complejidad, como el orden en el que se requería hace el informe.

Los parámetros indican un rango de inicio y fin en orden alfabético para los doctores de los cuales se desea el informe.

El ABB contiene la información necesaria para el informe completo, nombre del doctor, la especialidad a la que pertenece y la cantidad de pacientes atendidos por el doctor.

En el caso de que se pida un informe para todos los doctores, el ABB permite que el informe sea realizado en O(d), siendo d la cantidad de doctores en el sistema, en el caso de que no se pidan mostrar demasiados doctores, se implementó un iterador in-order para el ABB con un inicio y un final, que permita realizar el informe en O(log d).