TRABAJO PRÁCTICO N°2

Alumnos: - Camilo Fábregas
- Mariano Fortunato Rossi

Número de grupo: Grupo N°12

Corrector: Franco Di María

OBJETIVOS DEL TP

El objetivo de este TP es el de actualizar el sistema de gestión de la clínica Zyxcba Inc, originalmente en COBOL, por uno más moderno en C. Para ello, debemos implementar un sistema de turnos y un sistema de generación de informes para la clínica. Dada la complejidad del problema, decidimos primero plantear las opciones disponibles en papel y luego, una vez que se dispone de un plan de acción, empezar a programar nuestra solución.

Como primer paso, decidimos esbozar un boceto donde intentamos dividir los tres objetivos principales (los tres comandos), y dividirlos en problemas más pequeños. Consideramos muy importante elegir bien a priori las estructuras de datos a utilizar.

Como segundo paso, decidimos comenzar a implementar el sistema de una manera bastante "abstracta": pensamos las funciones auxiliares a utilizar, cómo subdividir el proyecto en varios archivos, etcétera. Llegamos a obtener un diseño general del proyecto, ya programado, que puede ser fácilmente editado en caso de tener cambiar alguna implementación por error de diseño.

Y como último paso, ya con un boceto hecho y una idea general de cómo llevarlo a código, implementamos nuestra solución usando los TDAs adecuados para cada situación.

BOCETO DEL PROYECTO

Trabajo Práctico 2

- **CSV**
  - Pacientes: nombre_paciente, año_inscripcion
  - Doctores: nombre_doctor, nombre_especialidad
  - Se devuelve una lista → Se podrían pasar a los pacientes a un **Hash**, y a los doctores a un **ABB**.
- **Especialidades y tipo de urgencia**
  - Urgencia: URGENTE ó REGULAR
  - Especialidad: Traumatología, Pediatría, Fisiatría, etc.
  - Los casos urgentes deberían ir a una **Cola**.
  - Los casos regulares deberían ir a un **Heap de máximos** (por antigüedad)
  - Los dos tipos de urgencia, y los tipos de especialidad podrían guardarse en dos **enum**. ✳
- **Comandos**
  - 1) Pedir turno: se recibe un paciente y una especialidad. Si es urgente, va a la cola de urgencias y si no, al heap.
  - 2) Atender paciente: se recibe un doctor, y si hay alguien con su misma especialidad en la cola de espera, se lo atiende. Sino, se atiende al primero del Heap con su especialidad.
  - 3) Imprimir informe: se imprime al doctor, especialidad y pacientes atendidos (contador en el ABB). Si se quiere un informe de varios médicos, se usa el sub-arbol que los contiene.

✳ Para los casos regulares, es mejor usar un **Hash de especialidades**, donde cada clave es una especialidad, y cada clave un **Heap de máximos** ordenado por antigüedad.

- El ABB de doctores (y no un Hash) sirve para tener un recorrido alfabético, muy útil para el informe. Esto sería con el recorrido **in-order**.
- **Escenarios posibles informe:**
  - Doctor1 = Doctor2 → Imprimo solo ese doctor
  - Doctor1 < Doctor2 → Imprimo rango de doctores
  - Doctor1 > Doctor2 → ERROR (rango no alfabético)
  - Doctor1; NULL → Imprimo desde el doctor1 hasta el final.
  - NULL; Doctor2 → Imprimo desde el inicio hasta el doctor2.
  - NULL; NULL → Imprimo todos los doctores.

Nuestro único cambio respecto al boceto inicial, es que decidimos crear un tipo de dato **especialidad_t**, que es un **struct** que contiene un Heap y una Cola. El Heap es de máximos y responde a los casos "regulares", que se ordenan de mayor a menor por el año de inscripción a la clínica. La Cola responde a los casos "urgentes", que son por orden de llegada. Entonces, cada especialidad (Kinesiología, Pediatría, etc) es una estructura que internamente contiene un Heap y una Cola. A su vez, todas las especialidades están guardadas en un Hash, lo que hace muy accesible la búsqueda de las mismas.

El uso de un **enum** para los tipos de urgencia fue descartado ya que su uso era irrelevante: simplemente se puede comparar la especialidad con la ingresada usando strcmp.

Todo lo demás fue implementado bajo ese boceto que creamos inicialmente. Esto fue muy importante para nosotros ya que de otra manera, se hubieran frecuentado mucho más los errores de diseño y/o de implementación.

ESTRUCTURAS UTILIZADAS

Para los pacientes, decidimos utilizar un **Hash**. El motivo es bastante trivial: solamente se tiene que guardar su nombre (una clave) y el año de inscripción a la clínica. Las búsquedas en Hash se reducen a O(1) con una buena función de hashing, por lo que no encontramos una alternativa que funcione mejor.

Para los doctores, decidimos utilizar un **ABB**. Si bien nuestra idea inicial era utilizar un Hash como para los pacientes, nos

dimos cuenta de que no sería posible conseguir un orden de O(d) (doctores) ya que se pide un informe en orden alfabético. Un ABB es muy útil para esto ya que con un recorrido in-order se puede obtener fácilmente el orden alfabético. El dato asociado de cada doctor es su especialidad, y la cantidad de pacientes atendidos.

Para las especialidades, optamos por un **Hash de structs** (tipo de dato especialidad_t), que tiene asociado un Heap y una Cola para cada especialidad. Esto ya fue explicado anteriormente, pero con esta implementación podemos conseguir un orden O(1) para pedir un turno urgente: se busca la especialidad en el Hash en O(1), se obtiene su cola de urgentes en O(1), y se encola al paciente en O(1). El heap de máximos es muy útil porque cada vez que se encola un paciente, este es ordenado automáticamente por su año de inscripción. Además, con el heap podemos mantener un orden O(log n) para el caso de los pacientes de urgencia regular para una especialidad específica.

Además, para el informe, decidimos utilizar un iterador externo de ABB para hacer el recorrido in-order. Esto nos sirve para poder tener, en una misma función, un recorrido que nos sirva tanto para contar la cantidad de médicos en un rango determinado, como para imprimirlos. Con un simple booleano, se puede diferenciar entre los dos usos que se le puede dar a ese recorrido.

COMANDOS: PEDIR TURNO

Nuestro algoritmo de **pedir turno** es muy simple. Primero chequeamos que el paciente esté en el sistema (hash de pacientes), luego chequeamos que la especialidad en la que se quiere atender exista en el sistema (hash_obtener distinto de NULL), y por último chequeamos que la urgencia sea de carácter "URGENTE" o "REGULAR". Si cualquiera de estas condiciones no se cumple, devolvemos.

Si la condición es urgente, a la especialidad obtenida con hash_obtener le pedimos su cola de urgentes con obtener_urgentes, y encolamos al paciente por su nombre.

Si la condición es regular, a la especialidad obtenida con hash_obtener le pedimos su heap de regulares con obtener_regulares. Luego creamos un struct paciente_t {nombre_paciente, anio_inscripcion}, y lo encolamos. Por su condición de heap de máximos, el sistema se encargará de darle su prioridad de acuerdo al año de inscripción.

COMANDOS: ATENDER PACIENTE

Para el comando de **atender a un paciente**, primero obtenemos al doctor asignado buscándolo en su ABB. Si el doctor no está en el sistema, devolvemos.

Luego obtenemos la especialidad del doctor, y buscamos la cola de urgentes y el heap de regulares de su especialidad, en el hash de especialidades. Si la cola tiene pacientes,

desencolamos uno de ellos (prioridad urgente). Si la cola está vacía y el heap tiene pacientes, desencolamos al de mayor antigüedad. Y si las dos están vacías, significa que no hay nadie esperando ser atendido para esa especialidad.

COMANDOS: INFORME DOCTORES

A la hora de imprimir los informes, y sabiendo que se reciben dos doctores por parámetro, primero analizamos qué caso será:

1. Doctor 1 = Doctor 2 → Imprimir solo ese doctor.
2. Doctor 1 < Doctor 2 → Imprimir rango de doctores.
3. Doctor 1 > Doctor 2 → ERROR (rango no alfabético).
4. Doctor 1 ; NULL → Imprimir rango Doctor 1 ; FINAL.
5. NULL ; Doctor 2 → Imprimir rango INICIO ; Doctor 2.
6. NULL ; NULL → Imprimir todo el árbol.

Primero revisamos el caso de ERROR, y el caso de que no haya doctores en el sistema (abb_cantidad = 0). Luego, lo que hacemos es iterar el árbol de doctores en el rango solicitado, y contamos la cantidad de doctores en dicho rango. Obtenido ese número, imprimimos por pantalla la cantidad de doctores que hay en el sistema para ese rango.

Luego iteramos una vez más, para el mismo rango, pero imprimiendo a cada doctor. Ambas iteraciones decidimos hacerlas con un iterador externo, utilizando condiciones de while y luego continuando el ciclo con abb_iter_in_avanzar.

Por ejemplo: para el caso 2. Empezamos en el comienzo del árbol, e iteramos hasta encontrar el doctor 1 (mientras el strcmp entre el actual y el doctor sea != 0, avanzamos con el

iterador). Una vez lo encontramos, desde ese punto hasta que se encuentre al doctor 2, hacemos:

- Sumamos +1 al contador de doctores (si queremos contarlos).
- Imprimimos al doctor (si queremos imprimir el listado)

Una vez que se llegó al doctor 2, no se necesita seguir iterando, por lo que se sale del ciclo. Una vez fuera del ciclo, destruimos el iterador y devolvemos.

MODIFICACIONES A TDAs

Para este TP, decidimos agregar dos primitivas muy simples a dos TDAs que desarrollamos a lo largo del cuatrimestre.

La primera primitiva que agregamos fue la de **cola_cantidad**, ya que nos es de enorme utilidad a la hora de querer imprimir los pacientes en espera cuando se le da el turno a un paciente. Simplemente es agregarle un contador al struct de la cola, que hace + 1 o -1 cuando se encola o desencola, respectivamente.

La segunda y última primitiva que agregamos fue **abb_in_order_por_rango** que, como su nombre lo indica, permite iterar el ABB desde un rango de inicio a uno final. Esto fue muy importante para el informe de los doctores.