Departamento de Computación
Algoritmos y Programación II

Número de Trabajo Práctico: 2

Docente a cargo: Martín Buchwald

Número de curso: 2

Ayudante: Lautaro Manzano

Número de grupo: 15

Fecha de entrega: 12/02/2021

Integrantes:

Apellido y Nombre	Padrón
Sabella Rosa, Cristóbal	106440
Sardella, Florencia	105717

Presentación del problema

El objetivo de este informe es analizar y detallar nuestra solución al problema planteado. El mismo hace referencia a una clínica médica en donde es necesario reemplazar la implementación del sistema de gestión encargado de organizar las listas de espera de los pacientes. Es esencial que los pacientes estén divididos por especialidad, dentro de ellas que los pacientes sean atendidos según un sistema de prioridades. Este último permite que aquellos pacientes catalogados como urgentes puedan ser atendidos con anterioridad sobre aquellos pacientes con estado regular. Los pacientes con consultas de urgencia serán atendidos por orden de llegada y con lo que respecta a los demás pacientes con consultas médicas se priorizará a aquellos que tengan mayor antigüedad en los registros de inscripción de la clínica. Conforme a esto nuestra meta es poder diseñar de la mejor y más eficiente manera posible un sistema que sea capaz de gestionar los turnos, generar informes de la clínica, y que mientras lleve a cabo estas acciones, cumpla con todas las condiciones nombradas previamente.

Con este fin, se nos provee de dos archivos en formato csv que especifican, en uno de ellos, una lista con los nombres de los pacientes y su año de inscripción en la clínica, y en el otro, los nombres de los doctores que trabajan en la clínica junto con la especialidad a la que se dedican. Una vez procesados los archivos y habiendo guardado toda la información necesaria que los mismos nos brindan, el sistema comienza a interactuar con el usuario, leyendo los comandos ingresados y efectuando las acciones que corresponden a cada uno de ellos. Estos comandos son tres.

El primero de ellos es el que se encarga de **pedir turnos**. Para ello se necesita el nombre del paciente solicitante junto con el nombre de la especialidad para la cual quiera una consulta y su grado de urgencia, catalogado como URGENTE o REGULAR. Una vez recibidos estos datos el sistema se encarga de poner en la lista de espera de dicha especialidad al paciente, siempre teniendo en cuenta que tuvo que haberse registrado en la clínica previamente (apareciendo en el CSV). Mediante la consola se informa al usuario que el paciente fue encolado y se detalla la cantidad de personas que están esperando ser atendidos para la especialidad solicitada en caso de ser efectiva. De lo contrario, el sistema proveerá un mensaje que indica si tanto el paciente, la especialidad o la prioridad asignada no existen. Este comando debe cumplir con las siguientes especificaciones:

- Si el paciente debe tratarse con urgencia, debe funcionar en O(1)
- Si es un caso regular, puede funcionar en O(log n) siendo n la cantidad de pacientes encolados en esa especialidad.

El segundo es el de **atender al siguiente paciente**, corresponde a cuando un doctor queda libre y por lo tanto puede atender a otro paciente. Acompañando este comando está el nombre del doctor. El sistema se encarga de buscar en los registros el nombre recibido y así poder identificar a qué especialidad se dedica con el fin de revisar su lista de espera. En caso de esta última no estar vacía, se devuelve el nombre del próximo paciente a ser atendido (teniendo en cuenta que se prioriza a los pacientes con urgencia por orden de llegada) y luego a los demás pacientes (según su año de inscripción en la clínica, dato tomado y guardado del archivo csv). En caso de ser exitoso, el sistema informa que se atendió al paciente e indica la cantidad de

pacientes que aún prevalecen en la lista de espera (si los hay) o que no hay pacientes, en caso contrario. Si el comando es ingresado con el nombre de un doctor que no existe, se avisará al usuario. Este comando debe funcionar de la siguiente manera:

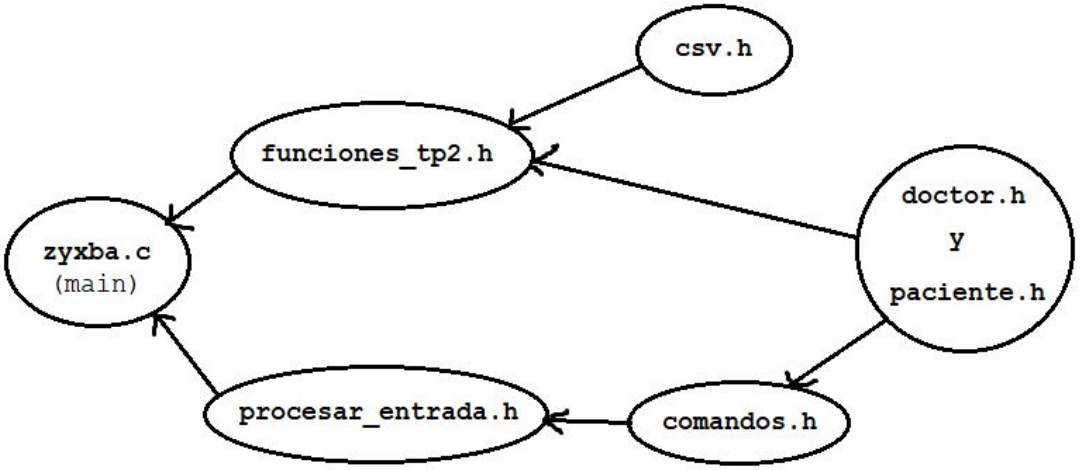
- Si el siguiente caso a tratar en la especialidad del doctor es urgente debe funcionar en O(log d), siendo d la cantidad de doctores en el sistema
- Si se trata de un caso regular debe funcionar en O(log d + log n), siendo n la cantidad de pacientes encolados en esa especialidad.

El tercer y último comando corresponde a la generación y posterior impresión de un **informe** que detalla en orden alfabético los nombres de los médicos que se encuentran en el sistema, con su especialidad y el número de pacientes que atendieron. Este comando brinda la posibilidad al usuario de especificar un rango de doctores sobre los cuales se desea un informe, si no es lo preciso simplemente se deja vacío y se tendrán en cuenta a todos los doctores. Este comando debe cumplir con:

- Debe funcionar en O(d) en el peor caso, siendo d la cantidad de doctores en el sistema (en caso de mostrar a todos los doctores)
- Debe funcionar en O(log d) en caso promedio, cuando no se pidan mostrar demasiados doctores.

Modularización

Debido a la cantidad de acciones que el sistema debe llevar a cabo se decidió separarlo en módulos para un mayor entendimiento y claridad. De esta manera en caso de ser apropiada una actualización se podrá acceder con facilidad al código específico, reduciendo los costos de tiempo e incrementando la efectividad del programa.
La estructura de los módulos utilizada se puede mostrar fácilmente con un grafo:



Detallando por cada módulo:

- **doctor.h y paciente.h**

En estos módulos se definen las **estructuras** doctor y paciente, que serán utilizadas para representar a cada individuo con sus datos, junto a funciones para su creación y destrucción para su uso en memoria dinámica.

- **csv.h**

Módulo proporcionado por la cátedra para la lectura de los csv recibidos.

- **funciones_tp2.h**

En este módulo se encuentran todas las funciones que utiliza el programa previamente a la interacción con el usuario:
- leer los csv de doctores y de pacientes
- asignar los datos a estructuras/TDAs para su fácil y rápido acceso

- **comandos.h**

Este módulo es donde se encuentran las funciones que se corresponden a los 3 comandos que pueden ser ingresados por el usuario:

- `PEDIR_TURNO`
- `ATENDER_SIGUIENTE`
- `INFORME`

- **procesar_entrada.h**

Este módulo es el que presenta la función que se encarga de la interacción con el usuario y de traducir su entrada, adjudicándosela a las funciones correspondientes.

- **zyxba.c**

Se encarga de ejecutar las funciones definidas en los otros módulos, de tal manera que primero se guarden los datos de los csv, para luego interactuar con el usuario utilizándolos y finalmente se destruyan las estructuras utilizadas (para no ocasionar pérdidas de memoria).

Estructuras y tipos abstractos de datos utilizados

Doctor y Paciente

Son estructuras que permiten "encapsular" los datos que se poseen sobre cada individuo. Al ser estructuras, su uso está pensado para acceder fácilmente a los datos que viven en ellas sin problemas, con la ventaja de poder referirse a todo el conjunto a través de un único puntero (por ejemplo, pudiendo encolar a un paciente con su nombre y su fecha de inscripción).

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

Los datos se corresponden al nombre del doctor, su especialidad y la cantidad de pacientes que logró atender mientras el programa estuvo abierto. Para facilitar, se lo define como **doctor_t**.

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

Los datos se corresponden al nombre del paciente y su año de inscripción. Para facilitar, se lo define como **paciente_t**.

Ambas estructuras fueron manejadas de forma dinámica para poder acceder a ellas desde el contexto de cualquier función, y su rápido acceso a través de punteros.

TDA Hash

Hace referencia a un tipo abstracto de datos con la ventaja de que sus primitivas permiten efectuar acciones en tiempo O(1).
Este TDA permite guardar datos asociados a claves mediante las cuales se podrán hacer búsquedas en tiempo constante. De esta manera, utilizamos el hash para almacenar punteros a los pacientes, siendo estas estructuras, conteniendo el nombre y el año de inscripción del paciente a la clínica, a las cuales podemos acceder fácilmente ingresando únicamente el nombre del paciente. Asimismo, nos habilita poder realizar cambios o actualizaciones rápidamente, como también chequear su pertenencia en el sistema (en O(1)).

TDA Árbol Binario de búsqueda modificado

El TDA árbol binario de búsqueda representa una estructura de datos en forma de árbol donde hay nodos entrelazados que cumplen con las siguientes condiciones: pueden tener como máximo dos hijos, haciendo referencia a lo binario, y su hijo izquierdo debe ser menor que él, mientras que el derecho debe ser mayor.
Este TDA, a diferencia de otros, permite que haya varias formas de iterar a todos sus nodos gracias a su ramificación. Estas podrían ser inorder, preorder, postorder y en niveles.
Debido a esta cualidad nosotros utilizamos un árbol binario de búsqueda con el fin de crear los informes de los doctores. Para esta función hacemos uso de la forma de iteración in order que nos habilita poder recorrer el árbol en orden alfabético. Sin embargo, al brindar la posibilidad al usuario de que ingrese un rango de nombres para la impresión del informe, modificamos al TDA para agregar una primitiva que permita iterar in order de forma tal que comience y termine su recorrido acorde a los caracteres y tiempos establecidos.
Con el objetivo de hacer una búsqueda relativamente rápida, es decir, una búsqueda en tiempo logarítmico (O(log d)), empleamos el algoritmo de búsqueda binaria. Este algoritmo permite dividir los datos en dos para luego poder seleccionar con cuál de los lados seguir operando, así el problema en cuestión se va achicando progresivamente al descartar los datos que no cumplen con las condiciones necesarias. Esta búsqueda es empleada en la primitiva del árbol binario `void *abb_obtener(const abb_t *arbol, const char *clave)`, esto nos permite obtener un nodo dentro del árbol en O(log n), siendo n la cantidad de elementos. En nuestro caso específico, al ser un árbol de doctores, en la función **atender siguiente paciente**, hacemos uso de esta cualidad para, una vez recibido el nombre del doctor, recorrer el árbol hasta hallarlo y así poder saber a qué especialidad se dedica, demorando O(log d), siendo d la cantidad de doctores en el sistema.
Nuevamente, hacemos uso de la búsqueda binaria para el comando **crear informe** con el fin de hallar la mínima raíz que entre sus hijos incluya el nodo que hace referencia al desde que ingresa el usuario. De esta manera, iniciando desde la raíz del árbol decide si el nodo actual es mayor, menor o igual al carácter ingresado como comienzo por el usuario. En caso de ser menor se queda solo con la ramificación derecha de la raíz, habiendo así achicado el problema descartado toda la rama izquierda. En caso contrario, ese es el nodo por el cual se comenzará a hacer la iteración in order. Esta iteración recorre hasta lo máximo posible hacia la izquierda, luego se analiza al nodo en cuestión y finalmente se recorre la derecha. En este recorrido nuestra función se encarga de no tener en cuenta a aquellos nodos que sean menores al inicio, evitando recorrer la ramificación izquierda del nodo en caso de este ser menor o igual al inicio establecido. En cuanto al final que indica el usuario nuestro algoritmo deja de recorrer el árbol una vez alcanzado el primer nodo mayor a esta condición.
De esta manera, con la primitiva añadida al abb, `abb_in_order_inicio`, podemos crear y luego imprimir por consola el informe de los doctores, en tiempo lineal, en caso de que ningún doctor sea excluido, o en tiempo logarítmico si se aplican limitaciones.
Para el caso en el que no se requiera un parámetro de comienzo la función que se utiliza es la del abb sin modificación:

```
void abb_in_order(abb_t *arbol, visitar_t visitar, void *extra);
```

Esta primitiva del árbol binario de búsqueda recorre in order todos los nodos aplicandoles la función visitar a los datos hasta que esta devuelva false. La función visitar es empleada con el fin de cortar la iteración cuando se alcanza un nodo con un valor mayor al límite establecido, en caso de haber uno.
Para el caso contrario se utiliza la nueva primitiva implementada:

```
void abb_in_order_inicio(abb_t* arbol, visitar_t visitar, void*
extra, const char* inicio);
```

Esta halla la mínima raíz que entre sus hijos incluya el nodo que hace referencia al desde que ingresa el usuario mediante una búsqueda binaria con la función:

```
abb_nodo_t* buscar_padre(abb_t* arbol, abb_nodo_t* nodo, const
char* ini);
```

Una vez encontrado este nodo se hace una iteración inorder entre los parámetros establecidos con la función:

```
void  _abb_in_order_inicio(abb_nodo_t*  nodo,  visitar_t  visitar,
void* extra, bool *seguir_iterando, const char* ini);
```

TDA Multicolas modificado

TDA Multicolas (sin modificar)

El TDA Multicolas (en su estado puro) está ideado como una serie de colas que actúan de forma independiente. Cada una con una clave asociada, para permitir su encolado y desencolado, de tal manera que un usuario pueda agregar la cantidad de colas necesarias para satisfacer su necesidad. Es posible una implementación en la que, salvo la de destrucción, todas las primitivas funcionen en O(1), utilizando un hash de colas.

Como todo TDA en memoria dinámica, tiene primitivas de creación y destrucción. La creación puede recibir opcionalmente una función de destrucción:

```
multicolas_t* multicolas_crear(destruir_dato_t destruir)
void multicolas_destruir(multicolas_t* multicolas);
```

También presenta primitivas para el manejo de colas, su cantidad, pertenencia al TDA, etcétera:

```
bool agregar_cola(multicolas_t* multicolas, const char* clave)
cola_t* quitar_cola(multicolas_t* multicolas, const char* clave)
bool cola_pertenece(multicolas_t* multicolas, const char* clave);
size_t multicolas_cantidad(multicolas_t* multicolas);
bool multicolas_esta_vacio(multicolas_t* multicolas);
```

Y finalmente, primitivas para encolado y desencolado de elementos:

```
bool multicolas_encolar(multicolas_t* multicolas, const char*
clave, void* elemento);
void* multicolas_desencolar(multicolas_t* multicolas, const char*
clave);
```

```
bool multicolas_cola_esta_vacia(multicolas_t* multicolas, const
char* clave);
```

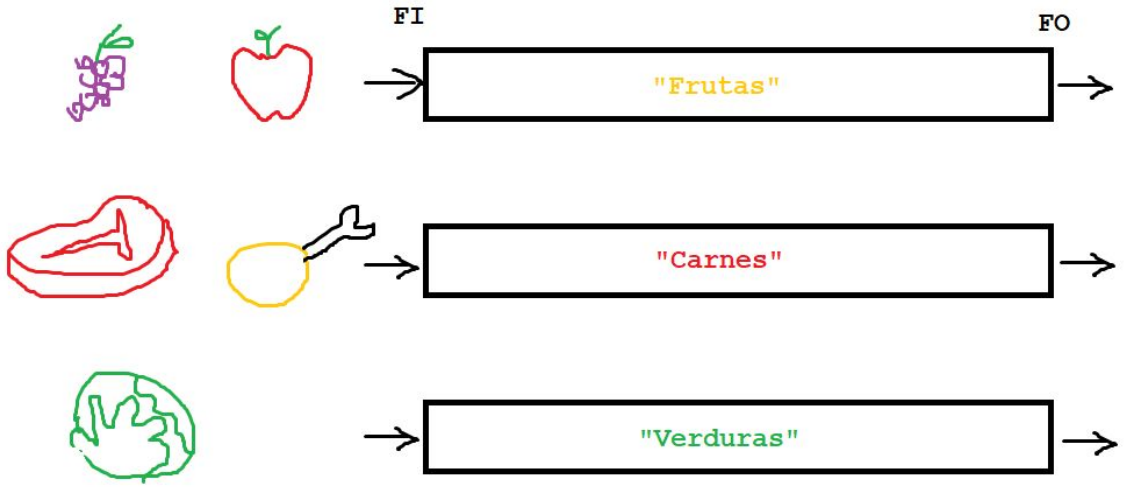
Ejemplo

Como ejemplo de uso, puede quererse encolar alimentos. Como se quiere diferenciar entre los tipo de producto, se pueden agregar colas para marcarlo. Entonces, ya habiendo creado la estructura, llamándola `alimentos`, se pueden crear colas con:

```
agregar_cola(alimentos, "frutas");
agregar_cola(alimentos, "carnes");
agregar_cola(alimentos, "verduras");
```

Entonces, se puede proceder a encolar cada elemento en su respectiva cola (cabe destacar que el TDA podría encolar una fruta en la cola "carnes" si así se especificara):

```
multicolas_encolar(alimentos, "frutas", manzana);
multicolas_encolar(alimentos, "carnes", pollo);
multicolas_encolar(alimentos, "frutas", uvas);
multicolas_encolar(alimentos, "verduras", repollo);
multicolas_encolar(alimentos, "carnes", bife);
```



Finalmente, se podría desencolar para cada categoría distinta, sabiendo que dentro de cada cola sólo se encolaron elementos correspondientes, respetando el orden First In-First Out para cada tipo de alimento específico:

```
multicolas_desencolar(alimentos, "frutas"); → manzana
multicolas_desencolar(alimentos, "frutas"); → uvas
multicolas_desencolar(alimentos, "frutas"); → cola vacía
multicolas_desencolar(alimentos, "carnes"); → pollo
multicolas_desencolar(alimentos, "verduras"); → repollo
multicolas_desencolar(alimentos, "carnes"); → bife
```

TDA Multicolas utilizado en la clínica

Como se puede prever, el TDA Multicolas en su estado original puede servir como un acercamiento a la estructura que necesitaría la clínica, pero no puede cumplir el rol completamente, principalmente debido al atendido basado en la fecha de inscripción (que no tiene por qué respetar el orden FIFO) y al sistema de urgencias. Es entonces que se propone utilizar un Multicolas modificado.
Esta modificación, bautizada TDA Multicolas Ordenadas con Prioridad, permitiría un Multicolas con colas que, en vez de respetar el FIFO, tengan un desencolado adaptado a las necesidades del problema. Estas colas, a las que nos referimos como TDA Colas de Prioridad, tienen la característica de actuar como un TDA Heap de Mínimos cuando un elemento no es prioritario (desencolado basado en orden), o un TDA Cola cuando sí lo es (FIFO). Es entonces que, para crear cada una de estas, necesitamos que se reciban dos funciones: una que determine si un elemento es prioritario o no (**func_priori_t**), y otra que compare entre elementos de la cola para establecer un orden (**cmp_func_t**). Además, para esta implementación particular, creamos una primitiva que permite ver la cantidad de elementos encolados dentro de la cola, para adecuarse a los requisitos del problema.
Esto se traduce, para el TDA Multicolas, en las modificaciones a las siguientes funciones:

```
multicolas_t* multicolas_crear(func_priori_t es_prioritario,
destruir_dato_t destruir)
```

La multicola ahora recibe una función para determinar prioridad al ser creada.

```
bool agregar_cola(multicolas_t* multicolas, const char* clave,
cmp_func_t cmp)
```

Al agregar una cola, se recibe la función de comparación que esta utilizará.

```
bool multicolas_encolar(multicolas_t* multicolas, const char*
clave, void* elemento, void* prioridad);
```

Al encolar, se recibe por parámetro un void* que servirá para determinar la prioridad del encolado (por ejemplo, en el caso puntual de la clínica, un string diciendo “URGENTE” o “REGULAR”).

```
size_t multicolas_cantidad_elementos(multicolas_t* multicolas,
const char* clave);
```

Además, para adecuarse a los requerimientos del problema, y aprovechando que las colas utilizadas permiten acceder a su cantidad de elementos, se creó una primitiva que, recibiendo una clave, permite ver la cantidad de encolados correspondientes.

Cabe mencionar que la primitiva `quitar_cola` no fue implementada, debido a que para la realización de la clínica no resultó necesaria, y es por eso que no se encuentra en esta versión particular del TDA.

Con este TDA, se pueden encolar y desencolar pacientes en la clínica en distintas colas (que representan las especialidades), respetando las directivas establecidas:

- Al encolar un paciente urgente, acceder a la cola de la especialidad sería O(1) y encolarlo también (por ser FIFO).
- Al encolar un paciente regular, acceder a la cola de la especialidad sería O(1) y, al actuar la cola utilizada como heap cuando no hay prioridad, el encolado sería O(log n), siendo n la cantidad de pacientes en esa especialidad.

- Al desencolar un paciente urgente, acceder a la cola sería O(1), y desencolarlo también (por ser FIFO). Sumado al O(log d) que tarda buscar al doctor correspondiente, el atendido respetará el tiempo establecido.
- Si se trata de un caso regular, será O(1) al buscar la especialidad, y O(log n) al desencolar, para que el siguiente atendido sea efectivamente el de mayor antigüedad en la clínica. Sumado al tiempo que tarda en buscar al doctor, quedaría O(log d + log n), cumpliendo con la consigna.

A las especialidades existentes se les adjudicarían colas dentro del TDA al procesar el CSV, lo cual permitiría también chequear rápidamente (O(1)) su existencia dentro del sistema.