# TRABAJO PRÁCTICO N°2

*Algoritmos y programacion II - Curso Martin Buchwald*

**Clínica Zyxcba Inc.**


## Grupo G22

## DELLA VECCHIA TOMÁS - 106368

## RIZZO EHRENBÖCK GONZALO DANIEL - 106475


Corrector designado: **Jasmina Sella Faena**

## Introducción:

El objetivo de este trabajo práctico es implementar un sistema el sistema de gestión básico de un hospital manejando las operaciones fundamentales de: Pedir Turno, Atender un paciente y obtener un informe de doctores.

Como primer paso (y antes de comenzar a escribir código) decidimos realizar un esquema de la solución, decidiendo y planificando qué estructura era la indicada para cada caso.

Nos ha ayudado para resolver el problema, pensar primero cómo funciona un hospital realmente (el manejo de colas, pacientes urgentes, doctores, informes, etc) y luego eso trasladarlo al código.

Con estas ideas ya planteadas, pasamos al código, cuyas especificaciones detallaremos a continuación.

# ESTRUCTURAS:

## Clínica:

```
struct clinica{
        abb_t *doctores;
        hash_t *especialidades;
        hash_t *pacientes;
        };
```

El struct clínica es la base fundamental de este trabajo, es la herramienta base de la cual depende todo el funcionamiento del programa.
Dentro de ella encontramos 3 estructuras (Dos hash y un árbol binario de búsqueda), que representan a los doctores, las especialidades de los doctores y los pacientes de la clínica.

Para la clínica, nos pareció lo más apropiado crear un TDA, el cual tiene las siguientes primitivas:

- -Clinica_crear
- -Obtener_doctores
- -Obtener_especialidades
- -Obtener_pacientes
- -Atender
- -Pedir_turno
- -Crear_informe
- -Clinica_destruir

## Doctores:

Decidimos representar cada doctor con el siguiente struct:

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

};
```

Para almacenar todos los doctores decidimos usar el tda ABB. Decidimos usar este por su rapidez de búsqueda, por la simplicidad que nos brinda para guardar datos en orden alfabético y así cumplir la complejidad de la función informe doctores y atender paciente.

## Pacientes:

Decidimos representar a los pacientes de esta manera:

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

Para almacenar los pacientes decidimos utilizar un hash. A comparación de los doctores, para los pacientes no precisamos en ningún momento listarlos en orden alfabético, simplemente precisamos saber sus nombres y su antigüedad en el sistema, para luego poder atenderlos. Además, saber si existe o no un paciente, es algo de complejidad constante en un hash, por lo que un hash nos pareció la mejor adaptación

a este problema

Especialidades:

Decidimos representar cada especialidad con el siguiente struct:

```
struct especialidad{
        char *nombre_especialidad;
        cola_t *cola_urgente;
        heap_t *cola_normal;
        size_t cantidad;

};
```

Notar que cada especialidad posee una cola urgente (para pacientes urgentes) y una cola normal (para pacientes sin urgencia).

Para la implementación de la cola urgente decidimos utilizar el tda Cola por su característica FIFO (first in first out), en la que se atiende al primero que llega.

Para la implementación de la cola normal decidimos utilizar el tda Heap. Una de las características de la clínica era que a sus pacientes normales los atiende por antigüedad. Es por eso que el Heap se vuelve fundamental en este caso ya que se atenderá al paciente de mayor antigüedad primero.

Para almacenar todas las especialidad decidimos usar el tda Hash. Decidimos usar este por su rapidez de búsqueda, búsqueda por clave y la no necesidad de almacenar las especialidades en orden.

**Salida y extra:**

```
typedef struct salida{
        char * nombre;
        char* especialidad;
        size_t atendidos;

}salida_t;


typedef struct extra{
        char** parametros;
        size_t contador;
        salida_t * salida;
        size_t tope;

}extra_t;
```

Estas estructuras son internas a la biblioteca de funciones_tp2.
Cabe destacar que estas estructuras se utilizan si, y sólo si, se pide un informe de los doctores con un rango alfabético específico.
Estas estructuras surgen por nuestra preferencia de no modificar ningún TDA previo.
Estas estructuras sirven para poder comunicarse con el iterador interno del abb y poder lograr completar el informe de doctores sin inconvenientes.
En sí, la estructura de salida, almacena en un vector toda la información de los doctores que es atravesada por el iterador interno del abb, para luego mostrarla por pantalla a modo de salida del programa.
Mientras que extra, es un parámetro extra, que en el caso de tener un rango limitado para mostrar un informe, en extra se almacenarán los rangos del informe y el string

que almacena toda la información de los doctores (Salida).

## FUNCIONES:

### Pedir Turno:

Si el paciente es urgente, se lo coloca en la cola de urgentes de esa especialidad (tda cola). La especialidad se obtiene del hash en tiempo constante O(1) y el encolado también es constante O(1). Se cumple con la complejidad constante.

Si el paciente es normal, se lo coloca en la cola normal (que es un tda heap).

La especialidad se obtiene del hash en tiempo constante O(1) (ya mencionado en párrafo anterior), como puede haber pacientes encolados en la especialidad indicada (n), la complejidad puede alcanzar O(log n) porque recordemos que la cola normal se ordena por antigüedad.

### Informe:

Esta función nos pide un listado de los doctores que existen en el programa, ese listado puede ser acotado, dependiendo de si así lo requiera quien la llame

En el caso de que se requiera un informe completo, se utiliza un iterador externo para el abb. Salvo que la cantidad de doctores en el árbol sea cero. De no ser así, se utiliza el iterador del árbol, recorriendo a este mismo de principio a fin, listando todos los doctores en su formato correspondiente.

En este caso, la complejidad tiende a ir a O(d), siendo d la cantidad de doctores en el sistema

Por otro lado, en el caso de que se requiera un informe parcial, se vuelve un poco más complejo el asunto y ahí es donde entran las estructuras extra_t y salida_t.
Para comenzar, en el caso del informe parcial siempre se usa el iterador interno. Lo que cambiará es la función booleana que este iterador recibe. Este cambio de función depende de cómo esté pedido el rango, si es desde un punto hasta el final, si es desde el comienzo a un punto o si es entre dos puntos.

A las funciones de visitado del iterador, se les pasará el parámetro extra, el cual contendrá un contador que contabilice los doctores encontrados en ese rango, como así también se pasará un vector del tipo salida_t, para almacenar toda la información de los doctores visitados y luego imprimirla por pantalla.

**Atender siguiente paciente:**

Primero se busca al doctor en el abb O(log d) (siendo d la cantidad de doctores en el sistema) y se busca en el hash la especialidad del doctor O(1)

Si el paciente a atender es urgente se desencola(1); quedando todo O(log d).

Si el paciente es normal, se desencola el heap O(log n) siendo n los pacientes normales ya encolados en la especialidad indicada; quedando todo O(log d+log n)