Informe TP2: Sistema de Generación de Turnos e Informes Clínicos

Tabla de Contenidos

	Estructuras de Datos	
Doctor	Paciente	Árbol de doctores
Hash de pacientes	Cola de Pacientes	Heap de Pacientes
Hash de turnos	Hash de turnos	Rangoe

	Código del programa	
main.c	load_structure_functions.c	command_functions.c
csv.c	error_messages.h	success_messages.h

Datos Personales del Grupo

Indice

Integrantes

- Millares Faustino.
- Litteri Iván.

Ayudante Asignado

- Ezequiel Genender

Análisis y Diseño (pre-escritura)

Fase de iniciación

Indice

1. Primera entrada del programa

El comando de ejecución del proyecto incluye 2 parámetros. Ambos son nombres de archivos CSV, el primero corresponde a el archivo CSV de doctores, el segundo corresponde a un archivo CSV de pacientes.

2. Manejo de errores de la primera entrada del programa

Se manejan los posibles errores que pueden llegar a aparecer, entre ellos: - que se reciban menos, o más de dos parámetros, - que alguno de los archivos no se puedan leer.

En caso de que alguno de estoy ocurra, la ejecución del programa se aborta con un exit con errno 1.

3. Lectura de CSV

Se leen los CSV, y se guardan las líneas leídas en una lista (una para cada archivo).

4. Manejo de errores de datos de archivos CSV

Hay que manejar los posibles errores que pueden llegar a aparecer luego de la lectura, que pueden deberse a: - algún año no es un valor numérico.

Fase de asignación de datos a estructuras

Indice

5. Almacenamiento de datos del archivo CSV de doctores

Se tiene que recorrer la lista de líneas del archivo de doctores, parsear a las mismas, y en este caso utilizaremos un struct para cada doctor, la que contendrá la información de su especialidad y la cantidad de pacientes que atendió.

Cada uno de estos doctores, se almacenará en una estructura de *árbol binario de búsqueda*, para cumplir con la complejidad pedida para el *comando informe doctores*, que debe ser *O(d)* en el peor caso (en el que se tenga que mostrar todos los doctores del sistema), *O(log d)* en un caso promedio (en el caso en el que no se pidan mostrar demasiados doctores). Y la estructura elegida cumple con esto. Las claves en dicha estructura corresponderán a el nombre de doctor, y su valor será la estructura mencionada en el párrafo anterior.

6. Almacenamiento de datos del archivo CSV de pacientes

Se tiene que recorrer la lista de líneas del archivo de pacientes, parsear a la misma, y en este caso no utilizaremos un struct para cada paciente, ya que la única información que disponemos y necesitaremos de los pacientes es su año de ingreso en la clínica.

Usaremos una estructura *hash* para almacenar los datos de los pacientes aprovechando su búsqueda de *O(1)* (para que no interfiera con la complejidad pedida en *pedir turno*). Donde la clave será el nombre del paciente y el valor el año de ingreso (a menos que necesitemos en algún momento la especialidad que requiere entonces pasaremos a una estructura).

De esta forma, cuando en *pedir turno* se acceda a estos datos, la búsqueda de pacientes será en *O(1)* y más adelante se explicará la estructura utilizada para guardar los turnos, la cuál (adelanto), involucrará una búsqueda en *O(1)* y un guardado en *O(1)* para casos *urgentes* y un guardado *O(log n)* para casos *regulares*.

Fase de comandos

Indice

El programa espera al ingreso de comandos. En esta instancia solo 3 tipos de comandos son válidos.

Uno que corresponde a el pedido de turnos. Este consta de 3 parámetros que debemos verificar que sea válido (paciente existente, especialidad disponible, y urgencia válida).

Vamos a usar un hash para los turnos urgentes y un hash para lo turnos regulares. Casa hash tendrá como clave la especialidad y cada especialidad como valor tendrá, una cola de espera ordenada por orden de llegada para los urgentes; y un heap ordenado por año de ingreso y orden de llegada en caso de tener el mismo año de ingreso para los regulares.

Fase de salida

Indice

Manejo de errores y devolución de la fase la buena fase.

Estructuras de datos

Doctor

Indice

Descripción

Esta estructura es la encargada de almacenar los datos de los doctores, los cuales son extraídos del archivo CSV correspondiente más un dato correspondiente a la cantidad de pacientes que antedió, que va a variar con la ejecución de los comandos (en específico con el comando `"ATENDER_SIGUIENTE"`).

Para su creación implementamos una primitiva que recibe como parámetros el nombre y la especialidad de un doctor (datos del CSV), para almacenar ésta información se pidió memoria exclusivamente en otro módulo, por lo tanto, como es responsabilidad del mismo liberar esa memoria, debimos realizar copias de estos datos para que la destrucción de esta estructura se responsabilice de sus datos sin interferir con los dependientes de otras estructuras.

También implementamos primitivas para obtener sus datos sin necesidad de acceder a la estructura interna, más una primitiva que incrementa el contador de pacientes atendidos por cada doctor.

Struct Doctor

```c
typedef struct
{
    char *name;
    char *specialty;
    int attended_patients;
} Doctor;
```

Primitivas Doctor

```c
/* Registra un doctor,
*  recibe el nombre
*  y la especialidad
*/
Doctor *doctor_check_in(char *name, char *specialty);

/* Obtiene el nombre del doctor
*  Pre: el doctor fue registrado
*  Post: devuelve nombre de doctor
*/
char *doctor_name(const Doctor *doctor);

/* Obtiene la especialidad del doctor
*  Pre: el doctor fue registrado
*  Post: devuelve la especialidad de doctor
*/
char *doctor_specialty(const Doctor *doctor);

/* Obtiene la cantidad de pacientes atendidos por el doctor
```

```
*  Pre: el doctor fue registrado
*  Post: devuelve la cantidad de pacientes atendidos por el doctor
*/
size_t doctor_attended_patients(const Doctor *doctor);

/* Suma 1 a la cantidad de pacientes atendidos por el doctor
*  Pre: el doctor fue registrado
*  Post: la cantidad de pacientes es atendidos aumento 1
*/
void doctor_attend_patient(Doctor *doctor);

/* Destruye los datos del doctor */
void doctor_destroy(Doctor *doctor);
```

Paciente

Indice

Descripción

Esta estructura es la encargada de almacenar los datos de los pacientes, los cuales son extraídos del archivo CSV correspondiente.

Para su creación implementamos una primitiva que recibe como parámetros el nombre y el año de entrada del paciente (brindados por el CSV); pedimos la memoria necesaria para almacenar la estructura, incluyendo memoria para una copia del dato del nombre, ya que la memoria que almacena a éste fué pedida en otra parte del código, la cuál es responsable por su liberación y al momento de la destrucción de la estructura no es nuestra responsabilidad liberar la memoria pedida en otro módulo (además causaría un error).

Además de las primitivas de creación y destrucción ya mencionadas implementamos las correspondientes al acceso de información que contiene la estructura, de esta manera nos aseguramos acceder a los datos de la estructura sin tocar implícitamente la estructura interna.

Struct Paciente

```
typedef struct
{
    char *name;
    int entry_year;
} Patient;
```

Primitivas Paciente

```
/* Registra un paciente */
Patient *patient_check_in(char *name, size_t year);
```

```c
/*  Obtiene el nombre de el paciente (no modifica al paciente)
*   Pre: el paciente fué registrado.
*   Pos: nombre del paciente.
*/
char *patient_name(const Patient *patient);

/*  Obtiene el anio de entrada del paciente (no modifica al paciente)
*   Pre: el paciente fué registrado.
*   Pos: anio de registro del paciente.
*/
size_t patient_entry_year(const Patient *patient);

/* Destruye los datos del paciente */
void destroy_patient(Patient *patient);
```

Árbol de doctores

Indice

Descripción

Esta estructura es un árbol binario de búsqueda autobalanceado (AVL) adaptado para que funcione como un árbol de doctores. En ella se almacenan las estructuras de los doctores, siendo la clave de cada nodo el nombre del doctor, y el valor la estructura Doctor con los datos asociados.

Para el desrrollo de esta estructura debimos implementar una estructura `AVL` implementado a partir del `abb` (mantiene sus primitivas, la diferencia está en las rotaciones entre punteros a nodos que "balancean" el árbol, mejora que es transparente al usuario). Al principio habíamos resuelto desarrollar esta estructura a partir del `abb` del que ya disponíamos, pero existía aún el caso *lista* que sucede cuando los elementos son ingresados al árbol en orden, se arma en forma de lista, lo que perjudica la complejidad de los comandos. Para resolver esto hay que hacer que el árbol se autobalancee. Entre todos los posibles, elejimos implementar el `AVL`. Otros motivo para elegir hacer un wrapper del `AVL` fué para manteter consistente el lenguaje que utilizamos para desarrollar el código, la idea de hacer una estructura con primitivas específicas para el caso, y además su coste temporal en búsqueda de datos cumple con lo necesitado.

Las primitivas son wrappers de las originales de `AVL`, y como se puede ver, no wrappeamos toda la estructura sino las primitivas que necesitamos. Debemos agregar que hemos hecho una modificación en `AVL`, implementamos una primitiva que itera al árbol por rangos internamente, debido a que no se nos ocurrió otra solución para recorrer el árbol en la complejidad pedida. Dicha primitiva funciona igual que el iterador interno original, sólo que recibe además dos parametros correspondientes al mínimo y máximo del rango de búsqueda requerido. De esta

forma implementamos una variante del algoritmo de *búsqueda binaria*.

Struct Árbol de Doctores

```
typedef AVL BSTDoctors;
```

Funciones Árbol de Doctores

```
typedef AVL_cmp_key bst_key_cmp;
typedef AVL_destroy_data bst_doctor_destroy;
```

Primitivas Árbol de Doctores

```
/* Crea la estructura */
BSTDoctors *bst_doctors_create(bst_key_cmp cmp, bst_doctor_destroy destroy_data);

/* Guarda un doctor en la estrucutra
*  Pre: la estrucutra fue creada, se registró al doctor.
*  Pos: se guarda al doctor en la estructura.
*/
bool bst_doctors_save_doctor(BSTDoctors *doctors, const char *doctor_name, Doctor *doctor);

/* Devuelve el struct doctor correspondiente al nombre de este.
*  Pre: la estrucutra fue creada, se registró al doctor y se guardó en la estructura.
*  Pos: se devulve el doctor correspondiente.
*/
Doctor *bst_doctors_get_doctor(const BSTDoctors *doctors, const char *doctor_name);

/* Devuelve el struct doctor correspondiente al nombre de este.
*  Pre: la estrucutra fue creada, se registró al doctor y se guardó en la estructura.
*  Pos: se devulve la cantidad de doctores registrados.
*/
size_t bst_doctors_count(BSTDoctors *doctors);

/* Destruye la estructura y sus datos si bst_doctor_destroy es distinto de NULL */
void bst_doctors_destroy(BSTDoctors *doctors);

/* Implementa el iterador interno, que recorre el arbol in-order.
   "visitar" es una función de callback que recibe la clave, el valor y un
   puntero extra, y devuelve true si se debe seguir iterando, false en caso
   contrario).
 */
void bst_doctors_in_order(BSTDoctors *doctors,
                  bool visit(const char *, void *, void *),
                  Rango *rango);
```

Hash de pacientes

Indice

Descripción

Esta estructura es un hash adaptado para que funcione como un hash de pacientes. En ella se almacenan las estructuras de los pacientes, siendo la clave de cada nodo el nombre del paciente, y el valor la estructura Paciente con los datos asociados.

La elección de implementar la estructura de esta forma, ya disponiendo de la estrucuta `hash` fué para manteter consistente el lenguaje que utilizamos para desarrollar el código, la idea de hacer una estructura con primitivas específicas para el caso, y además su coste temporal en búsqueda de datos cumple con lo necesitado. Para explicar más en detalle el último de los motivos, sabemos que la búsqueda y obtención de datos en un hash es *O(1)*, complejidad que no perjudicaría a las exigidas por la consigna en la ejecución de los comandos que necesiten acceder a esta estructura en búsqueda de sus datos.

Las primitivas son wrappers de las originales de `hash`, y como se puede ver, no wrappeamos toda la estructura sino las primitivas que necesitamos.

Struct Hash Pacientes

```
typedef hash_t HashPatients;
```

Primitivas Hash Pacientes

```
/* Crea la estructura */
HashPatients *hash_patients_create(hash_destroy_patient patient_destroy);

/*  Guarda un paciente en la estrucutra
*   Pre: la estrucutra fue creada, se registró al paciente.
*   Pos: devuelve si el paciente se registró correctamente.
*/
bool hash_patients_save(HashPatients *patients, Patient *patient);

/*  Informa si el paciente está registrado
*   Pre: la estrucutra fue creada, se registró al paciente.
*   Pos: devuelve si el paciente estaba registrado anteriormente.
*/
bool hash_patients_exists(const HashPatients *patients, const char *name);

/*  Devuelve un paciente
*   Pre: la estrucutra fue creada, se registró al paciente.
*/
Patient *hash_patients_get(const HashPatients *patients, const char *name);
```

```c
/* Destruye la estrucura y sus datos en caso de existir remanentes */
void hash_patients_destroy(HashPatients *patients);
```

Cola de Pacientes

Indice

Descripción

Esta estructura es la correspondiente a la cola de pacientes con turnos urgentes en espera. Su estructura corresponde a una cola en donde se encolan los pacientes urgentes por orden de llegada, y un contador que indica la cantidad de pacientes en espera en esa cola de espera. En la cola interna de la estructura se almacenan datos de tipo Paciente.

Para su implementación al principio pensabamos hacer un simple wrapper de `cola` que ya habíamos implementado, pero la estructura de dicha implementación no disponía de de un contador de datos encolados, entonces resolvimos agregar un contador que incrementa a medida que se encolan pacientes con turnos urgentes.

Las primitivas en su mayoría son wrappers de la estructura `cola`, salvo la primitiva encargada de su destrucción que además de destruir la estructura `cola` que forma parte de la estructura, destruye la estructura en sí (entiéndase destruir con liberar memoria). Como originalmente en la primitiva encargada de liberar la memoria de la cola se debe enviar una función de destrucción de datos, enviámos la función de destrucción de estructura Paciente. Otra modificación se hace en el momento de encolar un paciente, también se incrementa el dato de la cantidad. Y por último una primitiva encargada de informar dicho dato.

La elección de desta estructura se debe a que la complejidad a la hora de encolar y desencolar es constante ($O(1)$) por lo tanto cumple con la complejidad exigida.

Struct Cola de Pacientes

```c
typedef struct
{
    cola_t patients;
    size_t cant;
} QueuePatients;
```

Primitivas Cola de Pacientes

```c
/* Crea la estructura */
QueuePatients *queue_patients_create();

/*  Destruye la estrucutura .
*   Pre: la estructura fue creada.
```

```c
*/
void queue_patients_destroy(QueuePatients *patients, void (*patients_destroy)(void *));

/*  Informa si la cola se encuentra vacía (no modifica a la cola).
*   Pre: la estructura fue creada.
*/
bool queue_patients_is_empty(const QueuePatients *patients);

/*  Encola un paciente.
*   Pre: la estructura fue creada.
*/
bool queue_patients_enqueue(QueuePatients *patients, Patient *patient);

/*  Devuelve la información del primero en la cola, sin modificarla.
*   Pre: la estrucura fue creada.
*/
Patient *queue_patients_first(const QueuePatients *patients);

/*  Desencola un paciente.
*   Pre: la estructura fue creada.
*/
Patient *queue_patients_dequeue(QueuePatients *patients);

/*  Informa la cantidad de pacientes en espera.
*   Pre: la estructura fue creada.
*/
size_t queue_patients_count(QueuePatients *urgent);
```

Hash de turnos

Indice

Descripción

Esta estructura es la encargada de almacenar dentro de sí dos estructuras, ambas siendo hash, cada hash corresponde a una urgencia, por eso son dos. Cada hash tiene como claves a cada especialidad registrada a la hora que se registraron los doctores, y como valor, en caso del hash de turnos urgentes, una cola de pacientes, y para el caso del hash de turnos regulares, cada value corresponde a un heap de pacientes. Todos estos valores (claves, heaps y colas) se generaron vacíos luego de la fase inicial.

La estructura interna consta de 2 hash como se mencionón antes. La función primitiva que lo crea recibe como parámetros las funciones de destrucción de datos para cada uno de los hash que también se crean en ella.

Decidimos crear esta estructura para tener de forma prolija y en un solo lu-

gar (estructura) todo lo relacionado con el registro de turnos, además de que el comportamiento de la misma se justifica solo leyendo las primitivas que implementamos.

La primitiva encargada de liberar su memoria, destruye ambos hash por separado y luego la estructura en sí.

Struct Hash Turnos

```
typedef hash_t TurnsRegister;
```

Primitivas Hash Turnos

```
/* Crea la estructura */
TurnsRegister *turns_register_create(turns_register_destroy_data destroy_queue_patients,
                            turns_register_destroy_data destroy_heap_patients);

/*  Agrega un paciente a la cola de espera (dependiendo de la urgencia, va a
*   una u otra).
*   Pre: la estructura TurnsRegister fué creada.
*/
bool turns_register_add_turn(TurnsRegister *turns, char* urgency, char *specialty, Patient *

/*  Agrega una especialidad a el diccionario de turnos urgentes y regulares.
*   Pre: la estructura TurnsRegister fué creada.
*/
bool turns_register_add_specialty(TurnsRegister *turns, char *specialty);

/*  Atiende al siguiente paciente en espera.
*   Pre: la estructura TurnsRegister fué creada, la especialidad existe, y el
*       doctor está especializado en ella.
*   Pos: datos del paciente desencolado.
*/
Patient *turns_register_attend_patient(TurnsRegister *turns, Doctor *doctor, char *specialty

/*  Informa si existe una determinada especialidad.
*   Pre: la estructura TurnsRegister fué creada, la especialidad existe, y el doctor
*       está especializado en ella.
*   Pos: true si existe, false si no.
*/
bool turns_register_specialty_exists(TurnsRegister *turns, char *specialty);

/*  Informa la cantidad de pacientes en espera de una especialidad.
*   Pre: la estructura TurnsRegister fué creada.
*/
size_t turns_register_specialty_count(TurnsRegister *turns, char *specialty);
```

```c
/*  Destruye la estructura */
void turns_register_destroy(TurnsRegister *turns);
```

Heap de Pacientes

Indice

Descripción

Esta estructura es un heap adaptado para que funcione como un heap de pacientes. En ella se almacenan las estructuras de los pacientes con turnos regulares, almacenados por antigüedad.

La elección de implementar la estructura de esta forma, ya disponiendo de la estrucuta `heap` fué para manteter consistente el lenguaje que utilizamos para desarrollar el código, la idea de hacer una estructura con primitivas específicas para el caso, y además su coste temporal en búsqueda de datos cumple con lo necesitado. Para explicar más en detalle el último de los motivos, sabemos que la obtención de datos en un heap es *O(log n)*, complejidad que cumple con lo pedido para este tipo de turnos.

Como en la estructura guardamos punteros a estructuras `Paciente`, para ordenar por antigüedad a los mismos, implementamos una función de comparación que compara los años dentro de las estructuras de los pacientes entre ellos.

Struct Heap de Pacientes

```c
typedef heap_t Heap_Turns;
```

Primitivas Heap de Pacientes

```c
/*  Crea la estrucutra */
HeapPatients *heap_patients_create(HeapPatients_cmp cmp);

/*  Destruye la estrucutra, y si su función de destrucción no es NULL, también
*   sus datos.
*   Pre: la estructura fué creada.
*   Pos: se libera la memoria pedida para la estructura.
*/
void heap_patients_destroy(HeapPatients *turns, void (*patient_destroy) (void *));

/*  Informa la cantidad de pacientes en espera (no modifica la estructura).
*   Pre: la estructura fue creada.
*/
size_t heap_patients_count(const HeapPatients *turns);

/*  Encola un paciente.
*   Pre: la estructura fué creada.
```

```
*   Pos: devuelve si el paciente se encoló correctamente.
*/
bool heap_patients_enqueue(HeapPatients *turns, Patient *patient);

/*  Desencola un paciente.
*   Pre: la estructura fué creada.
*   Pos: devuelve el paciente desencolado.
*/
Patient *heap_patients_dequeue(HeapPatients *turns);
```

Rango

Indice

Descripción

Esta estructura almacena el rango de recorrido que se usa en el comando 3 para recorrer el árbol de doctores en rango. La estructura conforma dos punteros a arreglos de caracteres, que corresponden al nombre del doctor mínimo y máximo que limitan el recorrido, más un contador que va incrementando a medida que aparece un doctor que entra en el rango.

La creación de la estructura recibe los límites como parámetro, revisa las longitudes de los límites y si la longitud del límite mínimo es 0 (osea que no se indicó límite mínimo) se asigna la letra más chica respecto al código ASCII ("A"), lo mismo para el límite máximo solo que se asigna la letra más grande respecto al código ASCII ("z"); de otra forma se asignan copias de los mismos límites.

La destrucción de esta estructura se encarga de liberar la memoria de las copias de los límites y de la misma estructura.

Implementamos primitivas para acceder por medio de ellas a los datos que almacena sin acceder a la estructura interna directamente.

La elección de implementar esta estructura se debe a que, cuando pensamos en el funcionamiento del comando 3, pensamos en hacer un recorrido limitado (recorrido por rangos) en el árbol de doctores, pero como dicha primitiva recibe una función de visitar genérica, el dato que recibe como `void *extra` es uno sólo y debíamos trabajar con más de un dato.

Struct Rango

```
typedef struct Rango Rango;
```

Primitivas Rango

```
/* Crea la estructura */
Rango *rango_create(const char *min, const char *max);
```

```c
/*  Devuelve la cota inferior del recorrido del informe.
*   Pre: la estructura fué creada.
*   Pos: nombre o letra que marca límite inferior de la búsqueda, si es vacío,
*       se guarda "A" como mínimo caracter.
*/
char *rango_min(const Rango *rango);

/*  Devuelve la cota superior del recorrido del informe.
*   Pre: la estructura fué creada.
*   Pos: nombre o letra que marca límite superior de la búsqueda, si es vacío,
*       se guarda "z" como máximo caracter.
*/
char *rango_max(const Rango *rango);

/*  Informa la cantidad de doctores incluídos en el informe
*   Pre: la estructura fué creada.
*/
size_t rango_get_count(const Rango *rango);

/*  Incrementa en uno el contador interno de la estructura.
*   Pre: la estructura fué creada.
*/
void rango_count_increment(Rango *rango);

/* Destruye la estructura */
void rango_destroy(Rango *rango);
```

Código del programa

main.c

Indice

Descripción

Módulo principal que se encarga de ejecutar la función principal del programa haciendo uso de ambas librerías de funciones (`load_structure_functions.c` y `command_functions.c`).

Como es la encargada de crear las estructuras, también se hace responsable de la liberación de memoria de las mismas.

load_structure_functions.c

Indice

Descripción

Librería que incluye las funciones que cargan los datos de los archivos CSV en memoria.

Funciones

Indice

```
/*
*   Crea el esqueleto del diccionario de turnos urgentes y regulares utilizando
*   las especialidades que se encuentran en los datos de los doctores.
*   Pre: se creó la estructura doctor_csv_lines.
*   Pos: estructura del tipo TurnsRegister sin datos relevantes.
*/
TurnsRegister *load_turns_register(lista_t *doctor_csv_lines);

/*
*   Con los datos de los pacientes, crea estructuras para almacenarlos, y
*   almacena esas estructuras en un diccionario.
*   Pre: se creó la estructura patients_csv_lines.
*   Pos: estructura del tipo HashPatients con los datos de los pacientes cargados.
*/
HashPatients *load_patients(lista_t *patient_csv_lines);

/*
*   Con los datos de los doctores, crea estructuras para almacenarlos, y
*   almacena esas estructuras en un árbol binario.
*   Pre: se creó la estructura doctor_csv_lines.
*   Pos: estructura del tipo BSTDoctors con los datos de los doctores cargados.
*/
BSTDoctors *load_doctors(lista_t *doctor_csv_lines);
```

command_functions.c

Indice

Descripción

Librería que incluye a las funciones que se relacionan con la ejecución de los comandos del sistema.

Funciones

```
/*
*   Se recibe un nombre de paciente y el nombre de una especialidad, y el
*   sistema le añade a la lista de espera de la especialidad correspondiente.
*   Complejidad:
```

```
*       O(1) para casos urgentes porque, lo que hace en resúmen este
*   comando, es encolar el turno en la estructura correspondiente, en el caso de
*   los urgentes corresponde a una cola entonces encolar es constante.
*       O(log n) (siendo n la cantidad de pacientes en espera) para los casos
*   regulares porque, la estructura donde almacenamos los turnos regulares
*   corresponde a un heap, encolar es O(log n).
*       Como almacenamos a ambas "colas de espera" en un hash, entonces
*   acceder a ellas es constante, entonces no perjudica a la complejidad pedida.
*/
void make_appointment(TurnsRegister *turns, HashPatients *patients, char **parameters);

/*
*   Se recibe el nombre de le doctore que quedó libre, y este atiende al
*   siguiente paciente urgente (por orden de llegada). Si no hubiera ningún
*   paciente urgente, atiende al siguiente paciente con mayor antigüedad como
*   paciente en la clínica.
*   Complejidad:
*       O(log d) para los casos urgentes, la complejidad en este caso se la
*   lleva la búsqueda del doctor para confirmar su existencia, ya que como
*   almacenamos a los doctores en un árbol binario, su búsqueda cuesta O(log d)
*   (siendo d la cantidad de doctores). Y atender al paciente cuesta O(1) porque
*   los pacientes urgentes están encolados en una cola, y desencolarlos cuesta
*   eso.
*       O(log d + log n) para los casos regulares, la complejidad en este caso
*   es la de buscar a los doctores más la de atender pacientes, que en este caso
*   cuesta O(log r) siendo r la cantidad de pacientes regulares, ya que los
*   turnos regulares los almacenamos en un heap, y desencolar cuesta eso.
*/
void attend_patient(TurnsRegister *turns, BSTDoctors *doctors, char **parameters);

/*
*   El sistema imprime la lista de doctores en orden alfabético, junto con su
*   especialidad y el número de pacientes que atendieron desde que arrancó el
*   sistema. Opcionalmente, se puede especificar el rango (alfabético) de
*   doctores sobre los que se desean informes.
*   Pre: se creó y llenó de datos un registro de doctores.
*   Complejidad:
*       O(d) (siendo d la cantidad de doctores) en el peor de los casos ya que
*   se debe recorrer todo el árbol en caso de que el rango del informe lo pida.
*       O(log d) en el caso promedio, ya que no se recorre todo el árbol sino,
*   que se corta el recorrido al llegar al límite superior.
*/
void generate_rango(BSTDoctors *doctors, char **parameters);
```

csv.c

Indice

Descripción

Lee un archivo CSV, lo parsea y devuelve una estructura `lista` con las líneas del archivo, parseadas.

Funciones

```
/**
Haciendo uso de strutil (split) lee un archivo csv línea a línea.

Se devuelve una lista con todos los elementos construidos con split.
Es decir una lista de arrays (char**) donde cada array es una linea del csv.
devuelve NULL en caso de error al crear la lista.
**/
lista_t* csv_create(FILE* csv_file);

/**
 * Destruye la lista de arrays y el contenido de estos.
 **/
void csv_destroy(lista_t* list);
```

error_messages.h

Indice

Descripción

Librería de mensajes de error.

success_messages.h

Indice

Descripción

Librería de mensajes de ejecución de comandos exitoso e información.