# Trabajo Práctico Final - EDyAI

Valentina Prato

Diciembre 2020

## 1 Estructuras y Código

El primer obstáculo del trabajo fue definir una estructura adecuada para guardar los datos. Era necesario poder guardarlos de forma de poder buscar tanto por fechas como por localidades rápidamente, además de poder insertar o eliminar de forma simple. Las mejores opciones para esto eran arboles o tablas de hash, pero opté por la última ya que, con una buena forma de hasheo, resulta mucho más rápida y fácil de manejar para la mayoría de los comandos. Por lo tanto decidí usar una tabla de hash doble, hasheando primero por uno de los datos en la tabla principal para acceder una segunda tabla asociada a la clave, y en esta utilizar el segundo dato para finalmente llegar a los registros.

Luego, el siguiente obstáculo fue decidir la clave principal y secundaria. Algunos comandos, como buscar_pico, tiempo_duplicación y graficar, necesitan buscar registros de una localidad entre un rango de fechas, para lo cual convendría usar localidades como clave principal. Sin embargo también está imprimir_dataset, otro comando importante que requiere imprimir ordenando por fecha primero, luego departamento y lugar. En ambos casos hay comandos que necesitan saltar entre claves principales para la misma clave secundaria. Este problema, sin embargo, afecta mucho más a imprimir_dataset en el caso de guardar por localidades primero, ya que necesita acceder todos los registros guardados y debería saltar entre todas las localidades para cada fecha a imprimir, y para datasets como el de Santa Fe eso significa demasiado trabajo extra. En comparación, saltar entre fechas de un rango para solo una localidad es un costo muchísimo menor. Por lo tanto, elegí utilizar fechas como claves primarias y localidades como secundarias.

Algo importante a tener en cuenta sobre las localidades son los departamentos. Como mencioné antes, el dataset (luego de las fechas) está ordenado por departamento y luego localidad. Sin embargo consideré que agregar otro nivel de búsqueda sería una complicación innecesaria, ya que cada localidad se ingresa y se busca junto con su departamento. Una solución simple fue unirlos con coma para hacer una sola clave. De esta forma las claves secundarias de localidad tienen la forma de $Departamento, LOCALIDAD$ igual que están en el dataset (y así también se pueden imprimir de forma simple). Nótese además que solo la localidad está completamente en mayúsculas, esto no solo es el formato del dataset sino que es importante para diferenciar las partes cuando se ingresan como argumento separadas con espacio, ya que ambas pueden incluir espacios en si mismas. Cuando el programa quiere agregar un registro, es posible intentar separar las partes de "9 de Julio ANTONIO PINI" (para lo cual se definió la función marcar_lugar en straux.c), pero intentar diferenciar departamento y localidad de "9 de Julio Antonio Pini" es mucho más complicado o incluso imposible para un programa que no conoce el lugar. Por esta razón se pide al usuario diferenciar las partes de esa forma.

## 1.1 ltree

Luego de definir la estructura de doble tabla hash, surgieron dos problemas importantes relacionados a las localidades. Primero, cómo saber el orden en el que buscarlas a la hora de imprimir, ya que no se pueden hashear por orden alfabético. Segundo, el uso del espacio, ya que tener que alocar memoria nueva cada vez que se inserta la misma localidad en diferentes fechas es un gran desperdicio, sobre todo para datasets como el de Santa Fe.

Para matar dos pájaros de un tiro creé el árbol de lugares. Es un árbol de búsqueda binaria bastante simple para wide strings, con la función de guardar (por orden alfabético) cada localidad una vez y prestar el string (copiando el puntero) cada vez que se lo quiere insertar en la tabla. Incluye funciones de creación, inserción y destrucción, sin eliminación individual para evitar errores ya que el mismo string se comparte entre varios registros. La función inserción, por su parte, en lugar del wide string de localidad recibe un puntero al mismo (puntero a puntero a wchar_t) para poder reemplazar el string ingresado en caso de encontrarlo ya en el árbol. De esta forma, al ingresar un nuevo registro se intenta insertarlo al árbol primero, para que si ya pertenece se pueda liberar el nuevo string e insertar el registro con el preexistente. Este proceso realentiza la carga de datos ligeramente, pero ayuda mucho con la memoria y con la impresión del dataset.

## 1.2 hashf

La tabla de hash principal es la tabla Fechas, que utiliza punteros a estructuras de fecha (struct tm*) como claves asociadas a tablas Lugares y calcula la posición en la tabla como la diferencia en días respecto a una fecha de referencia. Esta fecha es la primera que se inserta, que se ubica en la posición 0 y se copia sobre el campo $ref$ de la tabla. Luego al insertar o buscar otras fechas se realiza $abs(dias(fecha, tabla->ref))$ para el hasheo. Elegí esa función de hasheo porque solo hay dos fechas posibles por casilla y se minimizan colisiones, además de que las fechas en general son consecutivas sin dejar demasiados espacios vacíos en medio a menos que se inserten fechas más alejadas individualmente. Y para evitar la inserción de fechas extremadamente lejanas, se limitan las fechas permitidas al realizar la conversión en string_fecha de straux.c a las posibles para registros válidos, es decir desde 2020 y no posteriores a la fecha del día.

En hash.c hay otras funciones auxiliares para los comandos como actualizar_fecha (copia una fecha sobre otra) y agregar_dias (que suma una cantidad positiva o negativa de dias a una fecha). También está la función igual_fechas para realizar comparaciones, pero la tabla no incluye campos $hash$ ni $igual$ porque el programa no necesita cambiar las funciones y resultaba innecesario.

Por último están además las funciones hash y paso para strings que utilizan las tablas de lugares, ya que se crean dentro en la funcion fechas_insertar.

Para realizar el hashing, se buscó utilizar una función simple que genere pocas colisiones para poder insertar y buscar rápidamente una gran cantidad de localidades. La función utilizada es una función polinomial rolling hash [1], que tiene la siguiente forma:

$$hash(s) = \sum_{i=0}^{n} s[i] \cdot p^i \ mod\, m$$

En esta función, $m$ es el tamaño de la tabla y $p$ un número primo generalmente cercano a la cantidad de letras diferentes en el string. Como el programa utiliza wchar_t, más grande que un char normal, puede guardar en un string cualquier caracter de ASCII extendido [2]. Y como debe poder aceptar cualquier caracter español, los cuales se encuentran bastante separados entre si en los valores extendidos (así que para diferenciarlos de otros caracteres y solo aceptar letras y números habría que hacer demasiadas comparaciones que consideré que no valdrían la pena), el programa acepta cualquier caracter dentro del string siempre y cuando mantenga el formato de "Departamento LOCALIDAD". Esto quiere decir que el string puede contener casi cualquiera de

los caracteres ASCII (sin contar los de control que estan en los primeros 31 y el 127), permitiendo 223 caracteres diferentes. Así que como 223 es un número primo, luego $p = 223$.

Por otro lado, como la cantidad de colisiones es inversamente proporcional al tamaño de la tabla, se suele usar un número primo grande para esto. Generalmente se recomienda $m = 10^9 + 9$, sin embargo al usar datasets como el de Santa Fe que tiene 352 localidades, es un tamaño que desperdicia demasiada memoria. Así que para tener un tamaño no tan extremo pero bastante mayor a la cantidad de localidades para minimizar colisiones, elegí $m = 997$.

La función de paso por el otro lado, simplemente retorna el valor ASCII del primer caracter. Esto también asegura que el tamaño no sea divisible por el valor de paso para (en el peor caso) recorrer toda la tabla antes de volver al primer lugar, ya que el máximo valor ASCII posible es 225 mientras que el tamaño de la tabla es 997, un número mayor no divisible por ningún número anterior.

## 1.3 hashl

La tabla de hash secundaria es la tabla Lugares, que utiliza wide strings como claves asociadas a punteros a enteros (para guardar los 3 enteros de casos confirmados, descartados y en estudio). Para la comparación utiliza simplemente !*wcscoll*, pero al igual que en la tabla Fechas no incluye campo *igual* para guardar la función ya que lo consideré innecesario para este programa.

Como los strings que se guardan son copias de los punteros guardados en el LTree, las funciones de esta tabla nunca los liberan, ya que esto se realiza desde el LTree al destruirlo.

## 1.4 straux

El archivo straux.c define mayormente funciones de manejo de strings. También incluye la función dias que maneja struct tm* ya que es necesaria para la verificación de fechas en string_fecha. Por esta función straux.h está incluido en hashf.h. El resto de las funciones se utilizan para procesar comandos en shell.c y comandos.c. Entre estos, la función más importante es wgetline, para leer lineas de stdin o de archivos. Es una versión de getline para wide strings ya que esta no existía y otras funciones como fgetws piden un limite de caracteres a leer, lo cual no era ideal.

## 1.5 comandos

El archivo comandos.c define las funciones sobre registros que se pueden llamar por comandos. Cada función actúa de la siguiente manera:

- **cargar_dataset:** Convierte el nombre del archivo a leer de formato wchar_t* (el formato en el que el programa lee de stdin) a char* para poder abrir el archivo. Si lo puede abrir, primero lee la primera línea para confirmar que haya datos y la descarta ya que debería tener los nombres de las columnas. Luego lee el resto del archivo línea por línea y toma tokens para separar y procesar las partes (asumiendo que las líneas tienen el formato AAAA-MM-DDT00:00:00-03:00,Departamento,LOCALIDAD,confirmados,descartados,enEstudio,total). Si todo se procesa correctamente, se inserta el lugar en el árbol (o si ya estaba se reemplaza el string por el del árbol para ahorrar memoria) y luego el registro se inserta en la tabla. Si no hay datos cargados previamente, la primer fecha leída se guarda como primer límite de fechas (ya que se asume que el dataset está ordenado por fecha primero) y la última que queda al terminar se guarda como último. De esta forma el orden de los límites reflejará el de las fechas en el dataset, lo cual ayuda a ordenarlas de la misma forma en imprimir_dataset. La idea de esto es que al imprimir luego de cargar sin modificar datos, los datasets de entrada y de salida

sean iguales (siempre y cuando todos los totales de la entrada sean correctos, lo cual no es el caso con el de Santa Fe). Si hay datos cargados antes del dataset, se mantiene el orden previo (si había una sola fecha se toma como ascendente).

El total se ignora al leer por ser innecesario para el programa así que no puede causar errores. Sin embargo, si hay error al procesar alguna de las otras partes, se detiene la lectura y se liberan todos los datos guardados en la tabla y en el árbol.

- **imprimir_dataset:** Convierte el nombre del archivo a escribir de formato wchar_t* (el formato en el que el programa lee de stdin) a char* para poder abrir el archivo. Comienza imprimiendo los nombres de columnas (Fecha,Departamento,Localidad,Confirmados,Descartados,En estudio,Notificaciones) y luego, si hay datos cargados en la tabla, imprime cada registro en el mismo formato en que cargar_dataset los lee, y ordenando por fecha y luego lugar (departamento y localidad, que se guardan unidos con coma). El orden de las fechas depende de el orden en el que se guardaron los límites al cargar los datos. Al acceder a cada fecha llama a una función auxiliar recursiva que recorre el árbol de forma inorder (para mantener el orden alfabético) buscando las localidades de los nodos en la tabla de lugares de la fecha para luego imprimir el registro si lo encuentra.

- **agregar_registro:** Recibe los argumentos a insertar en forma de strings. Luego intenta convertir el string de notificaciones a enteros y el de fecha a struct tm*. Si es necesario, actualiza las fechas limite. Finalmente, si se procesa todo sin problemas, busca el lugar en el árbol e inserta el registro en la tabla.

- **eliminar_registro:** Elimina el registro de la tabla si lo encuentra. Y si era el último registro para la fecha, libera la casilla de la fecha también. En este caso, de ser alguna de las fechas límites, reduce/aumenta el límite correspondiente por un día. El nuevo límite puede no ser exacto si el no tiene registros en la tabla, pero ahorra una lectura de la tabla para la mayoría de los comandos.

- **buscar_pico:** Busca todos los registros del lugar que recibe entre las fechas límites, empezando desde el límite anterior para recorrer la tabla entre las fechas posibles y calcular los casos confirmados diarios de cada registro que encuentra en búqueda de la mayor cantidad diaria. Cuando termina de recorrer todos sus registros, imprime el resultado.

- **casos_acumulados:** Simplemente busca e imprime el registro de casos confirmados en el lugar en la fecha, ya que los casos confirmados guardados son los acumulados hasta la fecha. Es un simple acceso a las casillas.

- **tiempo_duplicacion:** Busca el registro de casos confirmados en el lugar en la fecha y lo toma como referencia para luego recorrer las fechas hacia atrás en búqueda de la mitad o menos de casos. Recorre la tabla de la misma anera que buscar_registro pero en el sentido contrario. Se detiene al hallar el registro que lo cumple o al llegar al dia anterior al limite inferior de fechas.

- **graficar:** Compara el rango de fechas recibido con las fechas límite para intentar graficar todos los registros posibles en el rango ingresado. Una vez determinadas las fechas en las que se pueden encontrar registros, abre dos archivos: uno para guardar las fechas con sus casos diarios y otro para las fechas con los casos acumulados hasta el momento. Luego recorre la tabla entre las fechas (como buscar_pico y tiempo_duplicacion) e imprime todos los registros que halla. Si encuentra algún error en los datos donde la cantidad acumulada sea inferior a la última registrada, avista esto e imprime la acumulada como está pero imprime la diaria

como 0, para no graficar valores negativos. Al finalizar abre una instancia de gnuplot con popen (para utilizar esta función se define _GNU_SOURCE en comandos.h), imprime en ella todos los comandos necesarios y los ejecuta. Los gráficos se muestran uno debajo del otro en lugar de juntos ya que los valores acumulados suelen llegar a ser cientos de veces mayores a los diarios, y graficarlos con la misma escala haría difícil de ver estos últimos.

### 1.6 shell

El archivo principal es shell.c, donde se define el intérprete y el procesamiento inicial de comandos antes de pasarlos a sus respectivas funciones.

Al comenzar el programa se inicializan la tabla, el árbol y el array de fechas límite. Estas últimas, al igual que toda otra estructura de fecha, se inicializan con todos los campos en 0 utilizando memset. Esto se debe a que las estructuras struct tm* tienen muchos campos de los cuales el programa solo utiliza tres, y realizar opéraciones con estas estructuras sin inicializar todos los campos puede general errores (sobre todo en valgrind).

Luego se realiza la lectura de stdin en un loop en main, y si se ingresa algo aparte del enter (a menos que sea salir) el buffer se envía a la función procesar. Esta reconoce el comando y lo separa de los argumentos para también separarlos (a menos que sea solo un nombre de archivo) y pasarlos a sus respectivas funciones de comandos.c. En caso de haber varios argumentos marca las separaciones con —, a menos que sean departamento y localidad en cuyo caso marca con coma para después buscarlos como un solo string en las tablas.

El loop sigue hasta leer "salir", y luego libera todas las estructuras y termina el programa.

## 2 Compilación

A continuación se encuentran las opciones del Makefile para la compilación del programa:

- **make/make shell:** Compila el intérprete y todas sus dependencias (a menos que la última versión ya esté compilada).
- **make leaks:** Abre el intérprete con valgrind para chequear errores y/o pérdidas.
- **make clean:** Elimina todos los archivos .o (de la compilación) y .temp (creados al graficar).

Luego de compilar, el intérprete se puede ejecutar con ./shell.

## 3 Errores

Hay diferentes mensajes de error que pueden aparecer según el problema que encuentra el programa:

- **ERROR: Comando incorrecto:** El input no comenzaba con ninguno de los comandos permitidos.
- **ERROR: Faltan argumentos:** No se leyeron todos los argumentos que se esperaban.
- **ERROR: No hay registros de [fecha]:** La tabla no contiene registros para la fecha ingresada.
- **ERROR: Orden incorrecto de fechas:** Al ingresar las fechas para graficar, la primera debería ser anterior, pero es posterior.

- **ERROR: Fechas deben ser a partir del 2020:** La fecha ingresada es previa al 2020, y antes de eso no habían casos confirmados del virus (fuera de China).
- **ERROR: Fecha inválida:** La fecha ingresada no es una fecha válida del calendario.
- **ERROR: Fecha futura:** No pueden haber registros de fechas del futuro.
- **ERROR: Archivo vacío:** El archivo para cargar_dataset está vacío.
- **ERROR: Hubo un problema leyendo los datos:** No se pudieron leer correctamente los datos de la línea en cargar_dataset, probablemente no sigue el formato pedido.
- **ERROR: Numero invalido:** No se pudieron escanear correctamente los tres numeros esperados en agregar_registro.
- **ERROR: No hay registros cargados:** No se puede buscar en la tabla porque está vacía.
- **ERROR: No hay registros en [lugar]:** En buscar_pico no se encontró ningún registro de la localidad.
- **ERROR: No hay registros de [fecha]:** No se encontró la fecha en la tabla.
- **ERROR: No hay registros de [fecha] en [lugar]:** No se cargó ningún registro de la localidad en esa fecha.
- **ERROR: No hay registros en [lugar] con [n] o menos casos acumulados:** en tiempo_duplicacion no se puede calcular el tiempo porque no se registró la mitad o menos de casos que los de la fecha ingresada.
- **ERROR: No hay registros entre [fecha1] y [fecha2]:** El rango de fechas que se quiere graficar está completamente fuera de las fechas límites.
- **WARNING: Error de datos en [fecha]:** No es un error porque no detiene la función, pero avisa que en la fecha se registraron menos casos acumulados que los que había previamente.
- **WARNING: Solo hay registros desde/hasta [fecha]:** No es un error porque no detiene la función, pero avisa que no se pueden graficar registros de todas las fechas ingresadas.

# 4 Rendimiento

A continuación se encuentra una tabla con información extra sobre el tiempo aproximado (en segundos) que tardan los diferentes comandos para diferentes computadoras:

Se puede ver que la carga del dataset es lo que más tarda, posiblemente por buscar cada localidad de los aproximadamente 60 mil registros en el árbol, pero a pesar de esta pequeña demora el resto de las funciones son mucho más rápidas.

# References


[1] Hash de strings con rolling hash: `https://cp-algorithms.com/string/string-hashing.html#toc-tgt-6`

[2] Tabla ASCII: `http://www.asciitable.com/`

| **CPU** | i3-7100U | i5-7200U | i7-2600U | i7-7500U | Ryzen 7 3700x |
|---|---|---|---|---|---|
| **RAM** | 8GB | 8GB | 8GB | 8GB | 16GB |
| **cargar_dataset** | 1,5 | 1,2 | 1,3 | 1,3 | 0,8 |
| **imprimir_dataset** | 0,1 | 0,09 | 0,09 | 0,09 | 0,06 |
| **agregar_registro** | 0,00005 | 0,00004 | 0,00004 | 0,00004 | 0,00002 |
| **eliminar_registro** | 0,00002 | 0,00002 | 0,00002 | 0,00002 | 0,000008 |
| **buscar_pico** | 0,002 | 0,002 | 0,002 | 0,003 | 0,0009 |
| **casos_acumulados** | 0,000006 | 0,000006 | 0,00001 | 0,00001 | 0,000004 |
| **tiempo_duplicacion** | 0,00007 | 0,00005 | 0,0001 | 0,0001 | 0,00003 |
| **graficar** | 0,002 | 0,002 | 0,002 | 0,002 | 0,001 |