**Estructura de Datos y Análisis de Algoritmos**

**Laboratorio 1**

Kevin Arévalo Fernández

Profesor: Pablo Schwazenberg Riveros

Ayudantes: Diego Opazo

Javiera Saez

Javiera Torres

CONTENIDO

[CAPÍTULO 1. INTRODUCCIÓN 4](#_Toc515130659)

[1.1 OBJETIVOS 4](#_Toc515130660)

[1.1.1 Objetivo general 4](#_Toc515130661)

[1.1.2 Objetivos específicos 4](#_Toc515130662)

[CAPÍTULO 2. DESCRIPCIÓN DE LA SOLUCIÓN 5](#_Toc515130663)

[2.1 MARCO TEÓRICO 5](#_Toc515130664)

[2.1.1 Entradas 5](#_Toc515130665)

[2.1.2 Sobre estructuras de datos 5](#_Toc515130666)

[2.1.2.1 Arreglos 5](#_Toc515130667)

[2.1.2.2 Listas enlazadas 6](#_Toc515130668)

[2.1.2.3 Grafos 7](#_Toc515130669)

[2.2 HERRAMIENTAS Y TÉCNICAS 8](#_Toc515130670)

[2.2.1 Herramientas 8](#_Toc515130671)

[2.2.2 Técnicas 8](#_Toc515130672)

[2.2.2.1 Initialize-Single-Source(G, s) 9](#_Toc515130673)

[2.2.2.2 Relax(u, v, w) 9](#_Toc515130674)

[2.2.2.3 Dijsktra(G, w, s) 10](#_Toc515130675)

[2.3 ESTRUCTURAS DE DATOS 11](#_Toc515130676)

[2.3.1 Representación de un consultorio 11](#_Toc515130677)

[2.3.2 Representación de la red de consultorios 12](#_Toc515130678)

[2.3.3 Funciones y algoritmos 13](#_Toc515130679)

[2.3.3.1 Función buscar 14](#_Toc515130680)

[2.3.3.2 Función modificarCliente 14](#_Toc515130681)

[2.3.3.3 Función eliminarCliente 15](#_Toc515130682)

[2.3.3.4 Función agregarCliente 16](#_Toc515130683)

[2.3.3.4 Función generarReporte 16](#_Toc515130684)

[CAPÍTULO 3. ANÁLISIS DE RESULTADOS OBTENIDOS 17](#_Toc515130685)

[CAPÍTULO 4. CONCLUSIONES 18](#_Toc515130686)

[CAPÍTULO 5. REFERENCIAS 19](#_Toc515130687)

**ILUSTRACIONES**

[Ilustración 2‑1: Representación gráfica de un arreglo 6](#_Toc515131784)

[Ilustración 2‑2: Representación gráfica de una lista enlazada. 7](#_Toc515131785)

[Ilustración 2‑3: Ejemplo de grafo no dirigido y sin pesos. 7](#_Toc515131786)

[Ilustración 2‑4: Ejemplo de lista de adyacencia de un grafo no dirigido. 8](#_Toc515131787)

[Ilustración 2‑5: Función Initialize-Single-Source. 9](#_Toc515131788)

[Ilustración 2‑6: Función Relax. 10](#_Toc515131789)

[Ilustración 2‑7: Algoritmo de Dijsktra. 10](#_Toc515131790)

[Ilustración 2‑8: TDA Nodo. 11](#_Toc515131791)

[Ilustración 2‑9: TDA Grafo. 12](#_Toc515131792)

[Ilustración 2‑10: TDA ListaAdyacencia. 12](#_Toc515131793)

[Ilustración 2‑11: TDA NodoAdyacente. 12](#_Toc515131794)

[Ilustración 2‑12: Representación gráfica de la red de consultorios. 13](#_Toc515131795)

[Ilustración 2‑13: Pseudocódigo función buscar. 14](#_Toc515131796)

[Ilustración 2‑14: Pseudocódigo función modificarCliente solución 2. 14](#_Toc515131797)

[Ilustración 2‑15: Pseudocódigo eliminarCliente solución 1. 15](#_Toc515131798)

[Ilustración 2‑16: Pseudocódigo eliminarCliente solución 2. 15](#_Toc515131799)

[Ilustración 2‑17: Pseudocódigo agregarCliente solución 1. 16](#_Toc515131800)

[Ilustración 2‑18: Pseudocódigo agregarCliente solución 2. 16](#_Toc515131801)

[Ilustración 2‑19: Pseudocódigo generarReporte. 16](#_Toc515131802)

[Ilustración 3‑1: Menú principal de ambos programas. 17](#_Toc515131803)

**TABLAS**

[Tabla 2‑1: Funciones del programa. 9](#_Toc512787303)

# INTRODUCCIÓN

El Bulto Feliz, empresa de clínicas veterinarias conocida anteriormente, ha modernizado su sistema de registro de clientes mediante el software propuesto como solución y explicado en el Informe 1. Sin embargo, ha surgido un nuevo problema, esta vez relacionado con optimización de tiempo de traslado de pacientes a sus diversos consultorios distribuidos en distintos puntos de la región. El problema principal consiste en qué ruta escoger para derivar a un paciente desde un consultorio (que no posee la especialidad necesaria para realizar tratamiento) a otro que sí posea las herramientas, y como el tiempo es vital, por supuesto, esta ruta debe ser la más rápida. Además, el consultorio destino debe contar con un médico y espacio suficiente para atender al paciente.

Para solucionar el problema, se propone un software que sea capaz de leer la información de la red de consultorios, identificando todas las rutas disponibles entre ellos, especialidades y cupos, y mediante ésta, sea posible calcular la ruta más corta entre un consultorio origen (donde se encuentra el paciente) ingresado por el usuario, y un consultorio destino, que tenga la especialidad requerida (también especificada por el usuario) y el cupo suficiente.

1.1 OBJETIVOS

1.1.1 Objetivo general

* Diseñar y programar el software propuesto como solución al problema.

1.1.2 Objetivos específicos

* Desarrollar la solución utilizando grafos como estructura de datos principal.
* Lograr que el programa sea eficiente, en cuanto al tiempo de ejecución.
* Adquirir y/o mejorar habilidades de programación en el lenguaje C

# DESCRIPCIÓN DE LA SOLUCIÓN

2.1 MARCO TEÓRICO

Para resolver el problema de encontrar la ruta más corta entre dos consultorios dentro de una red, evidentemente, se necesita proporcionar al programa la información necesaria sobre la cual sea posible aplicar algoritmos resolutivos. Además de leer dicha información, es importante escoger con detalle, de qué forma ésta es almacenada y representada y luego, escoger una técnica eficiente para hallar la ruta más corta. Todos estos aspectos son abordados en el presente sub-capítulo y el siguiente.

2.1.1 Entradas

Como entradas al problema se consideran dos **archivos de texto plano**, que se definen como “aquellos formados exclusivamente por texto (sólo caracteres), sin ningún formato; es decir, no requieren ser interpretados para leerse (aunque pueden ser procesados en algunos casos). También son llamados archivos de texto llano, simple o sin formato”[[1]](#footnote-1). Uno de ellos posee información acerca de los consultorios (Nombre, especialidad, cupos) y el otro proporciona información acerca de cómo están conectados entre sí, especificando el tiempo que demora el trayecto entre uno y otro.

Para resolver el sub-problema de cómo representar adecuadamente una red de consultorios, cabe como anillo al dedo la utilización de la **estructura de datos** conocida como **grafo**.

2.1.2 Sobre estructuras de datos

También llamada TDA, es un tipo de dato compuesto en que cada elemento no es necesariamente del mismo tipo que los demás. Son útiles a la hora de tratar como un único dato a un conjunto de ellos que guardan cierta relación determinada.[[2]](#footnote-2) Ejemplos de ellos son los arreglos, las listas enlazadas y los grafos, que son, justamente las estructuras principales utilizadas para representar la información en el programa desarrollado. Antes de entrar en la definición de grafo, es necesario conocer las primeras dos estructuras mencionadas.

2.1.2.1 Arreglos

Un arreglo es una colección de variables del mismo tipo que se referencian por un nombre común.[[3]](#footnote-3) Los datos de un arreglo son fácilmente accesibles en el lenguaje C, ya que estos se referencian mediante un índice, que representa su posición en el arreglo. Existen arreglos estáticos: su tamaño no varía durante la ejecución del programa. Y también los hay dinámicos: su tamaño puede variar durante la ejecución del programa (aun así, el largo está explícitamente definido en todo momento).

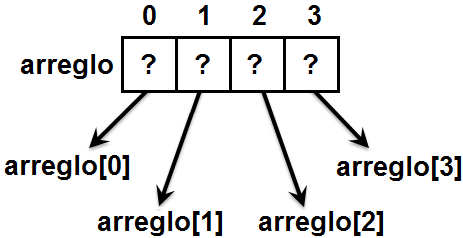


Ilustración ‑: Representación gráfica de un arreglo

2.1.2.2 Listas enlazadas

Una lista enlazada es un TDA similar a un arreglo, pero con la ventaja de que no es necesario saber, a priori, el número de elementos que va a tener la lista, ya que estos se van agregando dinámicamente según se necesite. A diferencia de los arreglos, sus elementos no están explícitamente indexados, por lo que es más difícil acceder a ellos, y es preciso crear funciones auxiliares con dicho propósito.

Una lista enlazada se compone básicamente de un nodo que contenga los datos que se desea almacenar (que puede ser cualquier TDA), y además posee necesariamente un puntero al siguiente nodo de la lista. El último elemento posee un puntero a NULL, para indicar que es el fin de la lista.

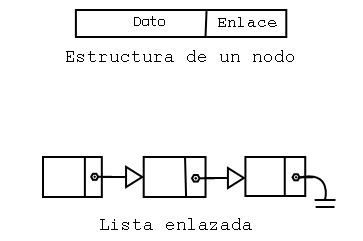


Ilustración ‑: Representación gráfica de una lista enlazada.[[4]](#footnote-4)

2.1.2.3 Grafos

Para las ciencias de la computación y la matemática, un grafo es una representación gráfica de diversos puntos que se conocen como nodos o vértices, los cuales se encuentran unidos a través de líneas o flechas que reciben el nombre de aristas.[[5]](#footnote-5) Existen grafos dirigidos y no dirigidos. En los grafos dirigidos, se especifica el sentido de las conexiones entre nodos, en cambio, en los grafos no dirigidos, esto no es así y se asume que, dada una conexión entre dos nodos, es posible llegar desde un nodo hacia el otro y viceversa. Además, un grafo puede poseer pesos en sus aristas, que son considerados como costos de llegar de un nodo a otro.

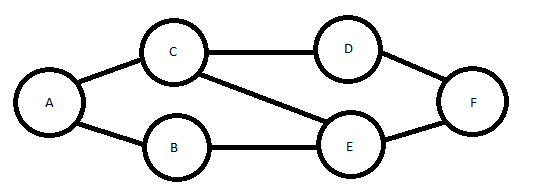


Ilustración ‑: Ejemplo de grafo no dirigido y sin pesos.

Para representar un grafo, existen dos formas muy utilizadas, que son mediante una matriz de adyacencia o una lista de adyacencia. En este caso se utiliza la lista de adyacencia, que consiste en una lista de todos los nodos del grafo, y cada nodo posee otra lista con todos los nodos adyacentes a él (de ahí el nombre).

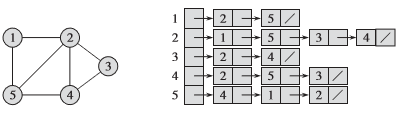


Ilustración ‑: Ejemplo de lista de adyacencia de un grafo no dirigido.[[6]](#footnote-6)

2.2 HERRAMIENTAS Y TÉCNICAS

2.2.1 Herramientas

Para llevar a cabo el proyecto, se utiliza el lenguaje de programación C, siguiendo el estándar ANSI-C para garantizar el correcto funcionamiento del programa en los sistemas operativos más utilizados. El compilador utilizado es GCC.

Se prioriza este lenguaje por sobre otros, debido al gran control que otorga a la hora de manejar la memoria del computador, lo que posibilita el desarrollo de un programa eficiente. Además, al ser tan estricto con los tipos de datos, se garantiza la correcta comprensión y aplicación de TDA’s por parte del programador.

2.2.2 Técnicas

La técnica principal en la que se basa el algoritmo que resuelve el problema de hallar la ruta más corta (entendiéndose la más corta como aquella que demande menos tiempo, no menos distancia) entre dos consultorios, es el Algoritmo de Dijkstra, que resuelve el problema de encontrar el camino más corto entre un nodo origen y todos los nodos de un grafo con pesos asociados entre sus aristas.

Antes de explicar el funcionamiento de dicho algoritmo, es necesario realizar una serie de definiciones de conceptos utilizados por este algoritmo para representar a un grafo y los caminos más cortos entre sus nodos.

Dado un grafo en donde es un conjunto de vértices o nodos, y es un conjunto de aristas (del inglés, edge) que poseen pesos o costos asociados entre ellas. Entonces, para cada vértice se mantiene un atributo , que representa al antecesor de , este atributo puede ser otro vértice o NULL. El atributo de cada vértice se establece de manera tal que la cadena de predecesores que se forma a partir de un vértice hasta llegar a un nodo , representa el camino más corto entre y .

El algoritmo de Dijsktra utiliza la técnica de **relajación**, que consiste en establecer para cada vértice un atributo , que mantiene el peso o costo total acumulado de llegar desde un nodo hasta el nodo , siguiendo además el camino más corto entre ellos. Se define como una **estimación del camino más corto**.

A continuación, se describen las funciones necesarias para aplicar el algoritmo.

2.2.2.1 Initialize-Single-Source(G, s)

Inicialización de los atributos , y . En donde es el nodo source o fuente

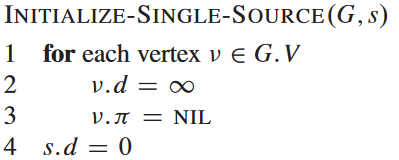


Ilustración ‑: Función Initialize-Single-Source.

2.2.2.2 Relax(u, v, w)

El proceso de relajar una arista consiste en determinar si es posible mejorar el camino más corto encontrado hasta el momento para llegar a , llegando a través de . Si este es el caso, se actualizan los atributos de . En donde w es el peso asociado entre y .

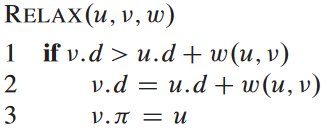


Ilustración ‑: Función Relax.

2.2.2.3 Dijsktra(G, w, s)

Finalmente, con todas las definiciones previas, es posible indicar cómo funciona el algoritmo:

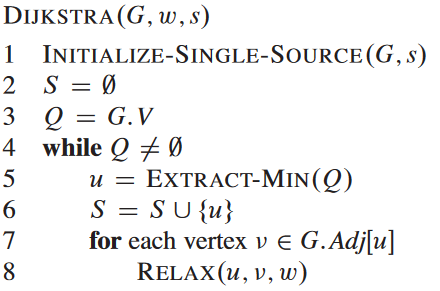


Ilustración ‑: Algoritmo de Dijsktra.

A medida que avanza el algoritmo, aquellos nodos a los que ya fue hallado su camino más corto, partiendo desde el nodo source, son agregados a la lista . El algoritmo, originalmente termina cuando se ha encontrado los caminos más cortos para todos los nodos pertenecientes al grafo. Eso será modificado para el caso particular de resolver el problema en tratamiento, pues el objetivo es encontrar el camino más corto hacia un solo consultorio destino, no a todos los de la red.

2.3 ESTRUCTURAS DE DATOS

En esta sección, se detallan las estructuras de datos utilizadas para formar el grafo que, eventualmente, represente a la red de consultorios de El Bulto Feliz.

2.3.1 Representación de un consultorio

Un consultorio se representa como un nodo, que formará parte de un grafo. El campo *padre* representa al atributo y *tiempoAcumulado* representa a la estimación del camino más corto. Además, debido a las características del problema, es necesario almacenar el nombre del consultorio, cantidad de pacientes atendidos actualmente en él, y su capacidad máxima de pacientes por atender.

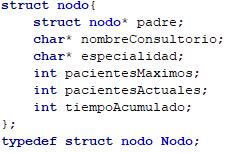


Ilustración ‑: TDA Nodo.

2.3.2 Representación de la red de consultorios

La red de consultorios es representada mediante un grafo no dirigido utilizando una lista de adyacencia. Para dicho efecto, se definen las siguientes estructuras.

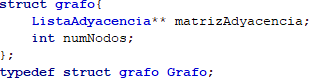


Ilustración ‑: TDA Grafo.

En donde *matrizAdyacencia* es una lista de *ListaAdyacencia*, estructura encargada de representar a un nodo (denominado origen) y contener una lista enlazada de nodos adyacentes a él, mediante otra estructura llamada *NodoAdyacente*, que se encarga de almacenar el nodo adyacente en cuestión y el costo (tiempo) entre él y el nodo origen.

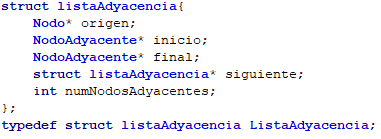


Ilustración ‑: TDA ListaAdyacencia.

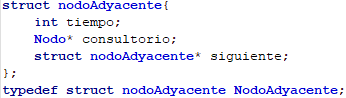


Ilustración ‑: TDA NodoAdyacente.

A continuación, se expone una representación gráfica de cómo están unidas estas estructuras, para una mejor comprensión de la abstracción del problema.

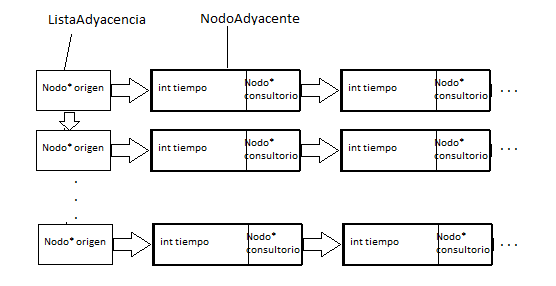


Ilustración ‑: Representación gráfica de la red de consultorios.

Cabe aclarar que el campo *consultorio* de cada *NodoAdyacente* es un puntero al consultorio adyacente en cuestión (permitiendo obtener las características propias del mismo), y no hacia el siguiente *NodoAdyacente* de *origen*. La estructura *Grafo* encapsula toda la información interpretada en la ilustración, y, además, almacena el número total de consultorios de la red (que es la misma cantidad de Listas de Adyacencia que posee *matrizAdyacencia*).

2.3.3 Funciones y algoritmos

A continuación, se muestran las funciones del programa. Son muy similares en ambas soluciones, incluso comparten sus entradas y salidas. En lo que sí varían (levemente) son en su complejidad.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Función | Entradas | Salida | T(n) 1 | O(n) 1 | T(n) 2 | O(n) 2 |
| cargarRegistro | char\* pathFile | DataBase\* | 16+36n | n | 18+40n | n |
| mostrarCliente | Mascota\* cliente | - | 10 | 0 | 10 | 0 |
| mostrarRegistro | DataBase\* db | - | 4+11n | n | 7+13n | n |
| crearCliente | - | Mascota\* | 30 | 0 | 31 | 0 |
| buscar | DataBase\* db, char\* query | - | 5n | n | 4+6n | n |
| modificarCliente | DataBase\* db, int i | - | 22 | 0 | 25+3n | n |
| eliminarCliente | DataBase\* db, int i | - | 8+5n | n | 9+3n | n |
| agregarCliente | DataBase\* db, Mascota\* cliente | - | 6+20n | n | 4 | 0 |
| volcarMascota | Mascota\* in, Mascota\* out | - | 19 | 0 | - | - |
| buscarPorIndice | DataBase\* db, int i | Mascota\* | - | - | 3+3n | n |
| freeMascota | Mascota\* mascota | - | 9 | 0 | 9 | 0 |
| generarReporte | DataBase\* db, char\* mes, char\* anio, char\* path | - | 7+22n | n | 8+23n | n |
| exportarRegistro | DataBase\* db, char\* path | - | 5+20n | n | 6+22n | n |
| liberarRegistro | DataBase db\* | - | 2+9n | n | 2+9n | n |

*Tabla 2‑1: Funciones del programa.*

Las funciones más relevantes, se detallan a continuación, mediante pseudocódigo. Si Existe un solo cuadro, quiere decir que la implementación es muy similar en ambos programas, sino, se explican por separado.

2.3.3.1 Función buscar

Para cada cliente en el registro:

Si query == nombreCliente o query = nombreMascota:

MostrarMascota(cliente)

Ilustración ‑: Pseudocódigo función buscar.

2.3.3.2 Función modificarCliente

Cliente\_a\_modificar = buscarPorIndice(db, i-1) //Función de orden n

Modificar datos según escoja el usuario

Ilustración ‑: Pseudocódigo función modificarCliente solución 2.

Para el caso de la solución uno (que utiliza arreglos), esta función se reduce a preguntarle al usuario qué cliente desea modificar, y se puede acceder a él rápidamente mediante la indexación.

2.3.3.3 Función eliminarCliente

NuevoArregloClientes[largo(db-arregloClientes)-1]

iterador = 0

j = 0

Para cada cliente en db – arregloClientes:

Si iterador == i – 1 // i representa el índice del cliente a partir de 1

Iterador++

Sino

NuevoArregloClientes[j] = db-arregloClientes[iterador]

iterador++

j++

vaciar(db - arregloClientes)

db – num\_Mascotas -= 1

db – arregloClientes = NuevoArregloClientes

Ilustración ‑: Pseudocódigo eliminarCliente solución 1.

Si i == 1

dliente\_a\_eliminar = db – inicio\_Lista

db – inicio\_Lista = db – inicio\_Lista – siguiente

FreeMascota(cliente\_a\_eliminar)

db – num\_Mascotas -= 1

sino

anterior = busscarPorIndice(db, i-2)

cliente\_a\_eliminar = anterior – siguiente

anterior – siguiente = cliente\_a\_eliminar – siguiente

freeMascota(cliente\_a\_eliminar)

db – num\_Mascotas -= 1

Ilustración ‑: Pseudocódigo eliminarCliente solución 2.

2.3.3.4 Función agregarCliente

NuevoArregloClientes[largo(db - arregloClientes) + 1]

Iterador = 0

Para cada mascota en db – arregloClientes

VolcarMascota(mascota, NuevoArregloClientes[iterador])

Iterador++

NuevoArregloClientes[iterador] = cliente

Vaciar(db - arregloClientes)

db – arregloClientes = NuevoArregloClientes

db – num\_Mascotas += 1

Ilustración ‑: Pseudocódigo agregarCliente solución 1.

Si db – inicio\_Lista == NULL

db – inicio\_Lista = cliente

db – final\_Lista = cliente

db – final\_lista – siguiente = NULL

sino

db – final\_Lista – siguiente = cliente

db – final\_Lista = cliente

db – final\_Lista – siguiente = NULL

db – num\_Mascotas += 1

Ilustración ‑: Pseudocódigo agregarCliente solución 2.

2.3.3.4 Función generarReporte

Archivo = fopen(“bultos.out”, “w”)

Para cada cliente en el registro

Si mesAtencion(cliente) == mes y anioAtencion(cliente) == anio

Escribir los datos del cliente en el archivo y un salto de línea

Ilustración ‑: Pseudocódigo generarReporte.

# ANÁLISIS DE RESULTADOS OBTENIDOS

El resultado obtenido satisface con el propósito de solucionar el problema, ya que se obtienen dos sistemas de manejo de clientes que funcionan y cumplen con las expectativas.

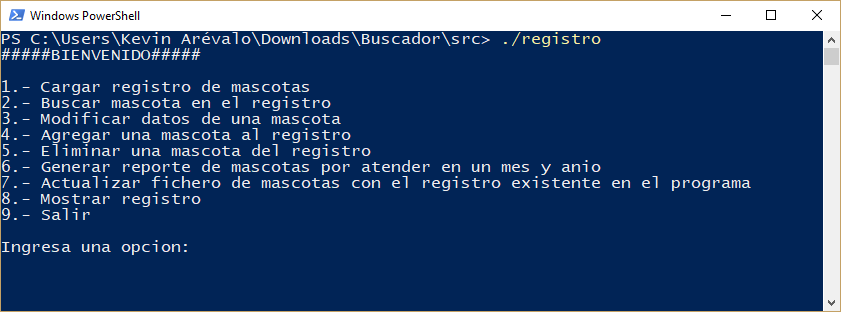


Ilustración ‑: Menú principal de ambos programas.

Con respecto a la comparación entre estructuras de datos empleadas, resalta el cambio de complejidad en algunas funciones, como, por ejemplo, en la función agregarCliente, que resulta ser una operación con complejidad constante si es que se utilizan listas enlazadas y se mantiene un puntero al final. En cambio, con arreglos, la operación tiene una complejidad de n, ya que primero se tiene que volcar la lista actual en una nueva con tamaño superior en una unidad, y finalmente agregar el elemento, que claramente tiene una complejidad de n. En el caso de la función buscarCliente, la balanza se inclina para los arreglos, pues basta con utilizar la indexación incorporada de los arreglos para acceder a un dato en específico, sin embargo, al utilizar listas enlazadas, esta operación requiere de una iteración para encontrar el dato.

El programa fue probado con un archivo de registro con 2 millones de clientes, y, en general, ha respondido de una manera aceptable: la carga del archivo no demora más de 5 segundos, y las operaciones de buscar, agregar, eliminar y modificar son casi instantáneas. Eso sí, al momento de mostrar al usuario la lista de clientes cargadas, se ocasionan problemas, al ser una cantidad muy elevada, la consola demora demasiado en mostrarlos.

Las falencias se ocasionan al ingresar datos solicitados por teclado: si éstos no cumplen con el formato especificado, el programa falla o se cae directamente.

# CONCLUSIONES

El objetivo principal: “Diseñar y programar el software propuesto como solución al problema”, se cumple, pues el programa obtenido posee las funcionalidades necesarias para cumplir con dicho propósito, y, además, se prueba con diferentes archivos, obteniendo los resultados deseados para cada uno de ellos.

Los objetivos secundarios también son cumplidos: se logra implementar la solución de dos maneras diferentes, utilizando arreglos y listas enlazadas. Durante el desarrollo de cada una de estas soluciones independientes, resaltan los pros y los contras de estas estructuras mencionadas, dejando una experiencia importante, pues, al conocerse sus características, se facilita la selección de una o la otra en proyectos futuros, no solo de la asignatura, sino del resto de la carrera y además de la vida laboral.

Indudablemente, el programa logrado no es perfecto, se puede agregar un sistema de verificación de entradas, tanto del archivo de texto, como de las entradas que ingresa el usuario por teclado. Además, en una situación real, sería necesario implementar una interfaz gráfica para una interacción más cómoda y eficiente con el usuario.

# REFERENCIAS

Calcifer.org. (2018). Estructuras de datos: listas enlazadas, pilas y colas. [online] Disponible en: http://www.calcifer.org/documentos/librognome/glib-lists-queues.html [Fecha de acceso: 20 de abril de 2018].

Köhler, J. (n.d.). APUNTES DE LA ASIGNATURA ANÁLISIS DE ALGORITMOS Y ESTRUCTURA DE DATOS. Santiago: Universidad de Santiago de Chile.

Alegsa, L. (2010). Definición de texto plano (archivo). Retrieved from <http://www.alegsa.com.ar/Dic/texto_plano.php>

Cormen, T. (2009). *Introduction to Algorithms* (3rd ed.). Massachusetts: Massachusetts Institute of Technology.

1. (Alegsa, 2010) [↑](#footnote-ref-1)
2. (Köhler, 2017) [↑](#footnote-ref-2)
3. (Schildt, 1999) [↑](#footnote-ref-3)
4. Imagen extraída de <http://www.calcifer.org/documentos/librognome/glib-lists-queues.html> [↑](#footnote-ref-4)
5. (Köhler, 2017) [↑](#footnote-ref-5)
6. (Cormen, 2009) [↑](#footnote-ref-6)